Tampereen yliopisto

Roope Pöyry

WEB-SOVELLUKSEN TIETOJEN TALLENNUS OFFLINE-TILASSA JA SYNKRONOINTI PALVELIMELLE

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2019

TIIVISTELMÄ

Roope Pöyry: Web-sovelluksen tietojen tallennus offline-tilassa ja synkronointi palvelimelle
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2019

Web-sovelluksia käytetään nykyaikana yhä enemmän kannettavilla laitteilla, kuten älypuhelimilla ja tableteilla. Tällöin joudutaan käyttämään usein mobiiliverkkoja, mikä voi tuottaa ongelmia esimerkiksi matkustettaessa tai syrjäisemmillä seuduilla liikuttaessa. Tällöin verkkoyhteydestä riippuvaisten web-sovellusten käytettävyys kärsii. Tässä työssä on tarkoitus tarkastella, kuinka tämä ongelma voidaan ratkaista offline-toiminnallisuuden ja tietojen synkronoinnin avulla. Tietojen synkronointi palvelimelle on tärkeässä osassa tukemassa offline-toiminnallisuutta, jotta muutokset saadaan myös palvelimelle. Web-sovelluksen offline-toiminnallisuuden mahdollistamiseksi täytyy selaimeen pystyä tallentamaan sekä sovelluksen vaatimia resursseja että sovelluksen käsittelemää dataa. Sovelluksen resurssit ovat useimmiten staattisia tiedostoja, joiden sisältö muuttuu harvoin. Sovelluksen data puolestaan muuttuu myös offline-käytön aikana, joten sitä pitää pystyä myös päivittämään offline-tilassa. Selaimeen täytyy siis tallentaa hyvin monenlaisia tietoja, joten tässä työssä tutkitaankin erilaisia vaihtoehtoja siihen tarkoitukseen.

Tässä diplomityössä toteutettu toiminnallisuus on tehty osana asiakasprojektia, jossa on tarkoituksena tarjota offline-toiminnallisuus maatiloille tehtävien työterveyshuollon tilakäyntien havaintojen kirjaamista varten. Tilakäyntien aikana tietoja voi muokata useampi henkilö, jolloin tietojen synkronoinnissa täytyy varautua myös mahdollisiin konfliktitilanteisiin. Koska maatilat voivat mahdollisesti sijaita hyvinkin syrjäisillä alueille maaseuduille, ei haluta olla pelkästään verkkoyhteyden varassa kirjauksia tehtäessä. Tästä syystä tilakännillä tehtävien havaintojen kirjaaminen on tarkoitus mahdollistaa myös offline-tilassa. Lisäksi tiedot täytyy synkronoida palvelimelle, jotta saadaan tarvittavat tiedot myös järjestelmään.

Lopputuloksena oli service workereita ja redux-offlinea hyödyntävä web-sovellus, jonka käyttö onnistuu ilman verkkoyhteyttä. Lisäksi tietojen synkronointi palvelimelle tapahtuu automaattisesti verkkoyhteyden ollessa jälleen saatavilla. Sovelluksen staattisten resurssien tallentaminen välimuistiin toteutettiin service workereilla. Näiden avulla sovelluksen suorittamiseen vaadittavat tiedostot saatiin helposti tallennettua pyyntö-vastaus-pareina selaimeen, josta niitä voidaan tarjota käyttäjän saataville, mikäli verkkoyhteyttä ei ole saatavilla. Sovelluksen datan tallentaminen ja tilan hallinta toteutettiin Redux-kirjaston avulla. Offline-toiminnallisuutta helpottamaan käytettiin lisäksi redux-offlinea, jonka avulla sovelluksen tilan tallentaminen paikallisesti localStorageen saatiin automatisoitua. Lopuksi tietojen synkronoinnissa toteutettiin yksinkertainen konfliktien havaitseminen ja ratkaisu siten, että viimeisimpänä tallennetut muutokset jäävät voimaan.

Avainsanat: Offline-toiminnallisuus, web-sovelluskehitys, service worker, datan synkronointi

ABSTRACT

Roope Pöyry: Saving data of web application in offline mode and synchronization to server
Master's Thesis
Tampere University
Information Technology
May 2019

Nowadays web applications are often used with portable devices, such as smart phones and tablets. Thus, mobile networks are needed, which may cause problems when traveling or in remote areas, where connectivity is poor. Because of this, the usability of network-based web applications suffers. The purpose of this thesis is to examine how this problem can be solved via offline functionality and data synchronization. The role of synchronizing the data with the server is important to support the offline functionality to get the changes to the server as well. The assets and the data of the application need to be stored into the browse to provide offline functionality. Application assets are often static files, which rarely change. The data in turn, changes during the offline usage, which means there must be a way to also update it in offline mode. Therefore, in this thesis multiple ways of storing resources into the browser will be examined.

The functionality in this thesis has been implemented as a part of a customer project, which means to provide offline functionality regarding to farm visits done in the system. The data of the visit can be modified by multiple users, which means possible conflicts must be taken into consideration. Because of farms' possible remote locations, it's not desirable to dependent of the network connection during visits. The observations done in visits can be saved in offline mode for that reason. Additionally, they will need to be synchronized to the server to get them to the system.

The result was web application which uses service workers and redux-offline to enable offline usage. The synchronization of data to the server happens automatically once network connection is available. Storing the static resources to cache was implemented using service workers, which makes it easy to store the required files into the cache as request-response-objects. They can be served to the user from the cache, when network is not available. Storing data of the application and managing application state was implemented with the Redux library. Additionally, redux-offline was used to help implementing the offline-functionality. It uses localStorage to store the application state locally automatically. Finally for the data synchronization, a simple conflict detection and resolution was implemented, where always the most recent changes are used.

Keywords: Offline functionality, web application development, Service worker, data synchronization

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

Kiitokset Hannu-Matti Järviselle ja Jerome Rannikolle ohjaajina toimimisesta ja palautteen antamisesta tiukankin aikataulun puitteissa. Lisäksi Adalia Oy:lle kuuluu kiitos mielekkään aiheen sekä työn toteuttamismahdollisuuden tarjoamisesta.

Tampereella, 21.5.2019

Roope Pöyry

SISÄLLYSLUETTELO

KUVALUETTELO

OHJELMALUETTELO

LYHENTEET JA MERKINNÄT

AJAX	engl. Asynchronous JavaScript And XML, asynkroninen tiedonvälitystapa selaimen ja palvelimen välille
API	engl. Application Programming Interface, ohjelmointirajapinta
Backend	Web-sovelluksen palvelinpuoli
CouchDB	Lokaalia tallennusta tukeva tietokanta
CSS	engl. Cascading Style Sheets, HTML-sivun tyylejä määrittävä tiedosto
Flux	Web-sovelluksille kehitetty arkkitehtuurimalli
Frontend	Web-sovelluksen käyttöliittymäpuoli
HTML	HyperText Markup Language, web-sovelluksissa käytetty tiedostomuoto
HTTP	HyperText Transfer Protocol, tiedonsiirtoprotokolla selaimen ja palvelimen välille
IE	Internet Explorer, verkkoselain
IndexedDB	Selaimessa tietojen tallennukseen käytetty tietokanta
JS	JavaScript, selaimessa usein käytetty ohjelmointikieli
JSON	engl. JavaScript Object Notation, tiedostomuoto tiedonvälitykseen
LocalForage	Useita eri tallennustapoja selaimessa tukeva rajapinta
LocalStorage	Selaimessa tietojen tallennukseen käytettävä rajapinta
Mela	Maatalousyrittäjien eläkelaitos
MVC	engl. Model-View-Controrller, Sovellusarkkitehtuuri
Offline-plugin	Offline-toiminnallisuutta tarjoava kirjasto, Webpackia käyttäville sovelluksille
PouchDB	Lokaalia tallennusta tukeva tietokanta
PWA	engl. Progressive Web Application, progressiivinen web-sovellus
Rails	Ruby on Rails, web-sovelluskehys
React	ReactJS, JavaScript-kirjasto käyttöliittymien rakentamiseen
Redux	Flux-arkkitehtuuria käyttävä kirjasto
Redux-offline	Offline-toiminnalissuutta toteuttava kirjasto Reduxin kanssa käytettäväksi
Redux-persist	Redux-sovelluksen tilan selaimeen tallentava kirjasto
REST	engl. Representional State Transfer, HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen
Service worker	Selaimessa taustalla suoritettava skripti
SPA	engl. Single-Page Application, yhden sivun web-sovellus
Turbolinks	Railsissa navigointia nopeuttava kirjasto

1. JOHDANTO

Web-sovelluksia käytetään nykyaikana yhä enemmän kannettavilla laitteilla, kuten älypuhelimilla ja tableteilla. Tällöin joudutaan turvautumaan langattomien verkkoyhteyksien, kuten mobiiliverkon, käyttöön kiinteän internet-yhteyden sijasta. Näiden verkkoyhteyksien kuuluvuus ja nopeus saattaa tuottaa ongelmia esimerkiksi matkustettaessa tai syrjäisemmillä seuduilla liikuttaessa. Tällöin sivujen latautumista joudutaan joko odottelemaan käyttäjän kannalta liian kauan, tai vaihtoehtoisesti näytetään käyttäjälle virheilmoitus verkkoyhteyden puuttumisesta, mikä tekee näistä sovelluksista käyttökelvottomia tällaisissa tilanteissa.

Tämä ongelma voidaan ratkaista tarjoamalla käyttäjälle mahdollisuus myös offline-tilassa työskentelyyn. Tässä työssä tarkoituksena on tutkia, kuinka mahdollistetaan websovelluksen käyttäminen offline-tilassa, sisältäen tietojen näyttämisen ja tallentamisen lisäksi myös synkronoinnin palvelimelle verkkoyhteyden ollessa jälleen käytettävissä. Tietojen synkronointi palvelimelle on tärkeässä osassa tukemassa offline-toiminnallisuutta kahdestakin syystä. Ensinnäkin käyttäjän muutosten synkronoitua palvelimelle hänellä on pääsy niihin myös muilta laitteilta, ja toiseksi muutokset saadaan näin myös muiden käyttäjän saataville.

Web-sovelluksen offline-toiminnallisuuden mahdollistamiseksi täytyy selaimeen pystyä tallentamaan sekä sovelluksen vaatimia resursseja että sovelluksen käsittelemää dataa. Sovelluksen resurssit ovat useimmiten staattisia tiedostoja, joiden sisältö muuttuu harvoin. Sovelluksen data puolestaan muuttuu myös offline-käytön aikana, joten sitä pitää pystyä myös päivittämään offline-tilassa. Selaimeen täytyy siis tallentaa hyvin monenlaisia tietoja, joten tässä työssä tutkitaankin erilaisia tapoja tietojen tallentamiseen selaimissa.

Tässä diplomityössä toteutettu toiminnallisuus on tehty osana asiakasprojektia, jossa on tarkoituksena tarjota offline-toiminnallisuus maatiloille suuntautuvien tilakäyntien suhteen. Tilakäyntien aikana tietoja voi muokata useampi henkilö, jolloin tietojen synkronoinnissa täytyy varautua myös mahdollisiin konfliktitilanteisiin. Koska maatilat voivat mahdollisesti sijaita hyvinkin syrjäisillä alueilla maaseuduilla, ei haluta olla pelkästään verkkoyhteyden varassa kirjauksia tehtäessä. Tästä syystä tilakäynnillä tehtävien havaintojen kirjaaminen on tarkoitus mahdollistaa myös offline-tilassa. Lisäksi tiedot täytyy synkronoida palvelimelle, jotta saadaan tarvittavat tiedot myös järjestelmään.

Tässä työssä käsitellään aluksi lyhyesti web-sovelluksia ja niiden arkkitehtuureja. Sen jälkeen tarkastellaan offline-toiminnallisuuteen liittyviä periaatteita sekä mahdollisia toteutustekniikoita. Luvussa 4 esitellään toteutuksen ympäristöön liittyviä tekijöitä, kuten sovelluksen arkkitehtuuria ja offline-toiminnallisuuden osuutta järjestelmässä. Lisäksi tarkastellaan hieman toteutuksessa käytettyjä teknologioita. Luvussa 5 puolestaan esitellään toteutukseen valitut ratkaisut sekä arvioidaan niitä ja mahdollisia vaihtoehtoisia tapoja. Lopuksi vedetään yhteen käsiteltyjä asioita sekä niistä saatuja tuloksia ja tehtyjä johtopäätöksiä.

2. WEB-SOVELLUKSET

Verkossa toimivat sovellukset eli web-sovellukset ovat selaimessa suoritettavia ohjelmia, jotka eivät vaadi ulkoisten liitännäisten asentamista käyttäjän koneelle. Nykyaikana niitä käytetään työpöytäkoneiden lisäksi kannettavilla laitteilla kuten älypuhelimilla ja tableteilla. Tästä johtuen sovellusten täytyy toimia hyvin erilaisissa selaimissa ja ympäristöissä. Tässä luvussa käsitellään web-sovellusten perustoimintaa asiakas-palvelin-mallin mukaisesti, nykypäivänä yhä yleisempiä yhden sivun sovelluksia sekä web-sovellusarkkitehtuureja.

2.1 Asiakas-palvelin-malli

Web-sovellukset noudattavat usein asiakas-palvelin-mallin mukaista arkkitehtuuria [1]. Tässä mallissa sovellus koostuu asiakkaasta eli verkkoselaimessa toimivasta käyttöliittymästä ja web-palvelimesta, jotka keskustelevat keskenään erillistä rajapintaa käyttäen. Asiakkaita voi olla useita, jotka tekevät pyyntöjä samalle palvelimelle, joka puolestaan reagoi näihin pyyntöihin [2]. Asiakas-palvelin-mallin avulla on eroteltu sovelluksen eri vastuualueita eri komponenteille esimerkiksi palvelintoiminnallisuuden yksinkertaistamiseksi ja skaalautuvuuden parantamiseksi.

2.1.1 Käyttöliittymä, palvelin ja rajapinta

Käyttöliittymä eli *frontend* tarkoittaa web-sovelluksen käyttäjälle näkyvää osaa eli se vastaa tässä mielessä aliluvussa 2.4.1 käsiteltävän MVC-mallin näkymää. Se sisältää käytännössä kaiken selaimeen lähetetyn tiedon eli usein HTML-dokumentin sekä mahdolliset tyyli- ja skriptitiedostot. Kun käyttäjä navigoi esimerkiksi johonkin toiseen polkuun, lähetetään pyyntö palvelinpuolelle eli *backendille*. Palvelimella pyyntö käsitellään ja haetaan tarvittavat tiedot tietokannasta ja lähetetään vastaus takaisin selaimeen. Palvelinpuolelle kuuluu kaikki sovelluslogiikka, mitä vaaditaan pyynnön käsittelyyn ja tietojen hakuun sekä mahdolliseen päivittämiseen. Lisäksi on tietokanta, jonka voidaan katsoa kuuluvaksi osaksi palvelinpuolta, vaikka se ei välttämättä sijaitse samalla palvelimella.

Käyttöliittymän ja palvelimen välisessä kommunikoinnissa käytettävä rajapinta eli API (*Application Programmin Interface*) on nykyään usein Fieldingin esittelemän [1] REST-arkkitehtuurin (*Representational State Transfer*) mukainen. Vaihtoehtoinen tapa tapa eli SOAP (*Simple Object Access Protocol*) on Tihomirovsin *et al.* mukaan [3] raskaampi ja

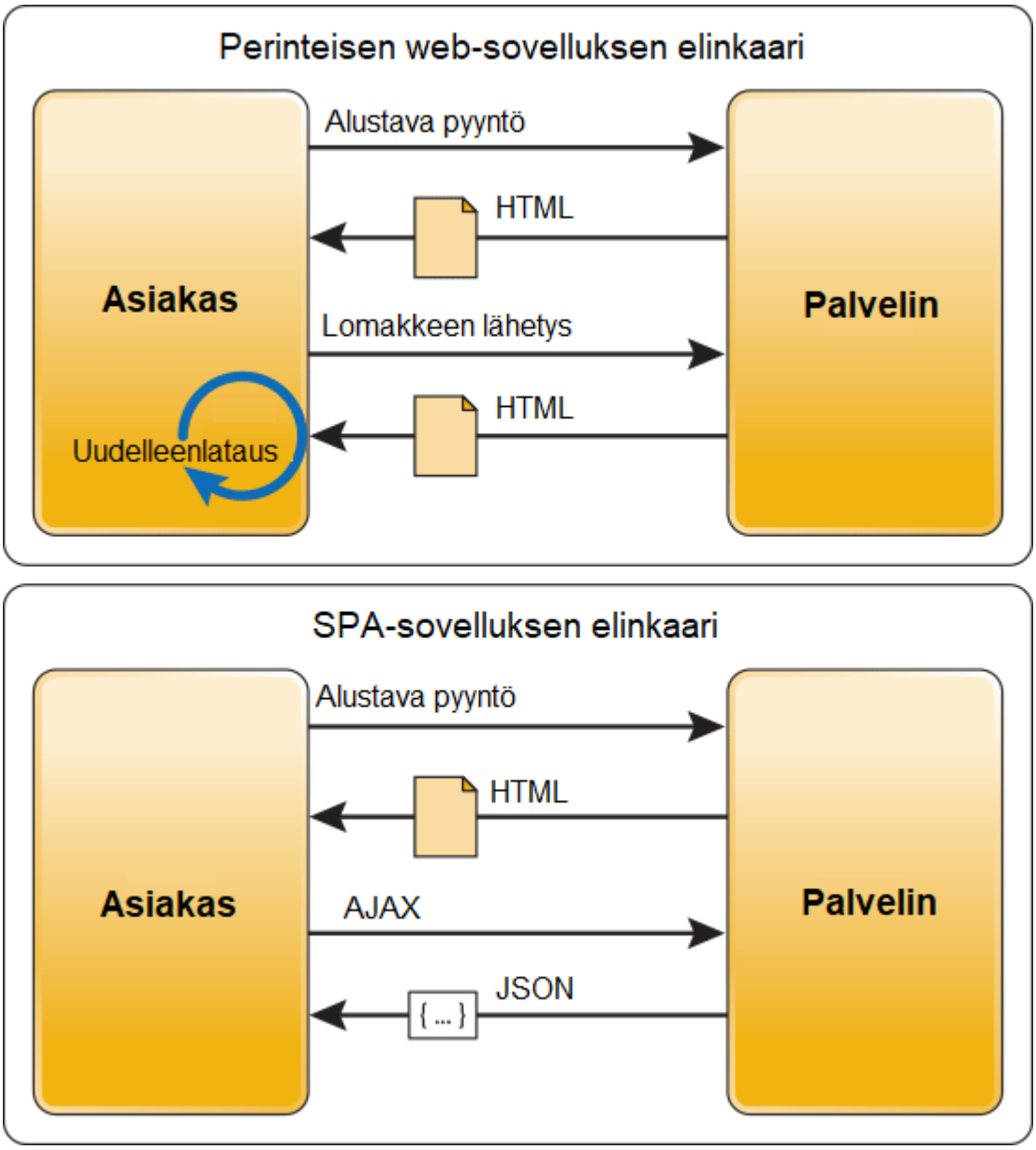
monimutkaisempi tapa, mutta sopii joihinkin esimerkiksi enemmän turvallisuutta vaativaan järjestelmään. REST-arkkitehtuurin kommunikointi on tilatonta eli yksittäisten pyyntöjen ymmärtämiseen ja suorittaminen pitää onnistua pyynnön mukana tulevien parametrien avulla. Tämä tarkoittaa käytännössä sitä, että käyttäjän istuntoon liittyvät tiedot välitetään jokaisen pyynnön mukana palvelimelle.

Tilattomuus yksinkertaistaa palvelimen sovelluslogiikkaa, sillä yksittäisiin pyyntöihin reagoidaan aina samalla tavalla riippumatta edellisistä pyynnöistä, mutta tämä toisaalta kasvattaa pyyntöjen mukana lähetettävää kuormaa. REST-rajapinnan kommunikaatio websovelluksen resursseihin voidaan suorittaa niin ikään tilattoman HTTP-protokollan avulla.

2.2 Yhden sivun sovellus

Nykyaikaisissa web-sovelluksessa ei enää välttämättä tarvitse ladata koko dokumenttia uudestaan, vaan esimerkiksi AJAX-pyynnöillä voidaan päivittää vain tarvittavat osat sivusta ja näin vähentää uudelleenlataamista ja parantaa käyttökokemusta. Tällaisia sovelluksia kutsutaan yhden sivun sovelluksiksi (engl. Single-Page Application). SPA-sovelluksissa ensimmäisen pyynnön jälkeen ei tarvitse enää ladata HTML-dokumenttia uudestaan, vaan ladataan vain tarvittavat tiedot esimerkiksi JSON-muodossa, joiden avulla voidaan päivittää dokumenttia. HTML:n lisäksi ensimmäisellä pyynnöllä ladataan kaikki tarvittavat JavaScript- ja CSS-tiedostot, joita tarvitaan sovelluksen käyttöliittymän toimintaan ja ulkoasuun.

Kuvassa 1 on esitetty, kuinka perinteiset web-sovellukset eroavat näistä ns. SPA -sovelluksista. Perinteisessä monen sivun sovelluksessa kaikkiin pyyntöihin vastataan HTML-dokumentilla, joka korvaa kokonaan edellisen dokumentin. Moderneista web-sovelluskehyksistä esimerkiksi Railsissa on versiosta 4 lähtien ollut oletuksena käytössä Turbolinks-gem, joka käytännössä tekee sovelluksista automaattisesti SPA-periaatteiden mukaan toimivan sovelluksen [4]. Turbolinks ei kuitenkaan tee sovelluksesta puhdasta yhden sivun sovellusta, sillä vastauksena voidaan silti lähettää myös HTML:ää. Turbolinks kuitenkin vain korvaa HTML-dokumentin body-elementin sisällön vastauksena saadulla uudella sisällöllä, jolloin koko dokumenttia ei tarvitse korvata nopeuttaen näin navigointia näkymien välillä.



Kuva 1. *Perinteisen ja SPA-sovelluksen ero lähteestä mukailtuna [5]*

Yhden sivun sovelluksissa selaimessa sijaitsevan toiminnallisuuden määrä kasvaa perinteiseen malliin verrattuna, kun näkymässä tapahtuvat muutokset renderöidään palvelimen sijasta selaimessa. Lisäksi ensimmäisellä latauksella tarvittavan käyttöliittymäkoodin määrä hidastaa ensimmäistä latausta. Yhtenä SPA-sovellusten huonona puolena Tilkov mainitsee [6] sen, että sovelluksen kaikkiin resursseihin navigointi ei onnistu polun avulla. Tämä ei välttämättä moderneissa yhden sivun sovelluksissa ole enää ongelma, jos selainpuolella on toteutettu reititys eri resursseihin. Hyötynä on toki se, että saadaan eriytettyä käyttöliittymään liittyvä toimintalogiikka palvelinpuolen sovelluslogiikasta [7]. Lisäksi navigointi nopeutuu, koska yksittäisillä pyynnöillä siirretään huomattavasti vähemmän dataa, mikä parantaa sovelluksen responsiivisuutta ja sitä kautta käyttökokemusta.

2.3 Web-sovellusarkkitehtuurit

Web-sovellusten kehitykseen ja niiden rakenteeseen liittyvät keskeisellä tavalla web-sovellusarkkitehtuurit. Arkkitehtuuri antaa suuntaviivat sovelluksen suunnitteluun ja toimintaan sekä helpottavat ratkaisemaan tunnettuja web-sovelluksille yhteisiä ongelmia. Tällöin sovelluskehittäjät voivat jättää sovellusten perustoiminnallisuuteen liittyvät sovelluskehyksen vastuulle ja keskittyä itse sovelluksen toimintalogiikkaan. Tässä käsitellään lyhyesti web-sovelluksissa suosittua MVC-arkkitehtuurimallia (*Model-View-Controller*) ja sen perustoiminnallisuutta sekä verrataan tähän Flux-arkkitehtuurimallin toimintaa ja tiedonkulkua sovelluksessa.

2.3.1 MVC- arkkitehtuurimalli

Krasner ja Pope määriittelivät 1980-luvun loppupuolella MVC-arkkitehtuurimallin. Se on noussut web-sovelluskehityksessä suosituksi arkkitehtuurimalliksi, jota käytetään edelleen monissa sovelluskehyksissä, kuten ASP.NET, Ruby on Rails ja Spring [8]. Pääperiaatteena MVC:ssä on sovelluslogiikan jakaminen kolmeen itsenäiseen moduuliin, jotka ovat:

1. malli (engl. *Model*)
2. näkymä (engl. *View*)
3. kontrolleri (engl. *Controller*).

Tämän avulla sovelluksen logiikka jaetaan erillisiin yksiköihin, joilla on omat tehtävänsä ja vastuualueensa. Tämän jaottelun avulla saadaan sovelluslogiikkaa yksinkertaistettua, sillä yhdessä komponentissa voidaan keskittyä toteuttamaan sille kuuluvaa tehtävää [9]. MVC:n soveltuvuus web-sovelluksiin perustuu siihen, että näkymän ja kontrollerin tehtävät jakautuvat luonnollisesti selaimen ja palvelimen välille. Selain toimii tiedon esittäjänä ja palvelin on vastuussa sovelluksen toimintalogiikasta [8].

Mallin tehtäviin kuuluu olioiden ja tietokannan välisen suhteen ylläpitäminen. Esimerkiksi SQL-tietokantojen tapauksessa yksi malli vastaa yleensä yhtä tietokannan taulua. Mallin kautta pystytään suorittamaan tietokantaan niin luku- kuin kirjoitusoperaatioitakin. Malli toimii siis kerroksena sovelluksen ja tietokannan välissä. Tämän lisäksi mallissa voidaan validoida tietoa ja luoda assosiaatioita eri mallien välille. Näkymä on vastuussa tiedon esittämisestä käyttäjälle ja toimii käyttöliittymänä, joka mahdollistaa käyttäjän ja sovelluksen välisen vuorovaikutuksen. Kontrolleri puolestaan on vastuussa sovelluksen toimintalogiikasta. Se käyttää mallia hakeakseen ja päivittääkseen dataa ja lähettää sen eteenpäin näkymälle esitettäväksi jossakin muodossa. [10]

Kuva 2. *MVC-arkkitehtuurimalli mukaillen lähteestä [11]*

Käytännössä nämä sovellukset toimivat kuvassa 2 esitetyn kaavion mukaisesti siten, että käyttäjän tekee pyynnön johonkin osoitteeseen. Kontrolleri käsittelee pyynnön ja hakee tietoja tarvittaessa mallilta, jonka jälkeen käyttäjälle näkyvää näkymää päivitetään sen mukaan. Ensimmäiselle kerralla vastaus on todennäköisesti HTML-dokumentti, joka mahdollisesti sisältää tai lataa lisäksi skriptejä tai tyylejä käyttöönsä. Tämän jälkeen sivua eli näkymää päivitetään käyttäjän syötteiden mukaan.

2.3.2 Flux-arkkitehtuurimalli

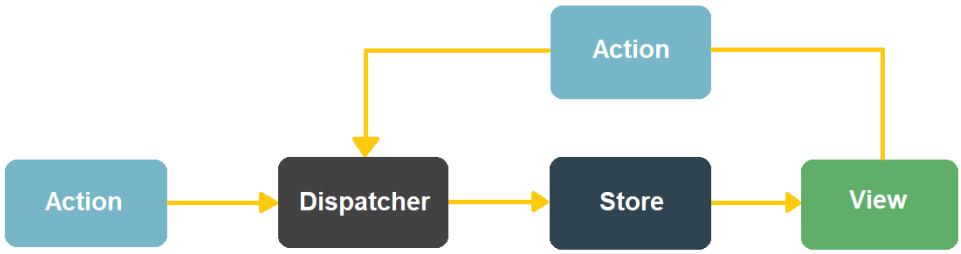
Flux on sovellusarkkitehtuuri, jonka tarkoituksena on välttää MVC-sovelluskehyksille tyypillinen monisuuntainen tiedonkulku. Se tarjoaa yksisuuntaisen tiedonkulun yksinkertaistamaan sovelluslogiikkaa. Tämän monimutkaistuminen voi olla MVC:ssä ongelmana, kun mallien ja näkymien määrä kasvaa. Useammassa näkymässä saatetaan tarvita samaa dataa, jolloin tiedonkulkuun tulee riippuvuuksia. [12], [13]

Flux-sovellukset koostuvat neljästä pääkomponentista:

1. *dispatcher*-komponentti
2. *store*-komponentti
3. *view*-komponentti
4. *action*-komponentti.

Nämä muistuttavat MVC:n komponentteja, mutta eivät ole sama asia, sillä esimerkiksi kontrollerit ovat Fluxissa kontrollerinäkymiä, jotka ovat sovellushierarkiassa ylimpänä. Ne noutavat dataa store-komponentilta ja välittävät tämän datan edelleen lapsikomponenteilleen. [14], [15]

Kun käyttäjä tekee näkymässä jonkin toiminnon, näkymästä lähtee toiminto dispatcher-komponentin kautta kaikille store-komponenteille, jotka sisältävät sovelluksen dataa ja toimintalogiikkaa. Nämä puolestaan päivittävät kaikkia näkymiä, joihin muutokset vaikuttivat. Tämän tyyppinen tiedonkulku toimii esimerkiksi ReactJS:n tyyppisissä deklaratiivissa ohjelmointikielissä, missä näkymien päivittämiseen ei tarvitse määritellä, kuinka siirtyä tilasta toiseen, vaan se on automatisoitu.



Kuva 3. *Tiedon kulku Flux-arkkitehtuurissa [14].*

Kuvassa 3 on kuvattu tiedonkulku Flux-sovelluksissa. Kuvassa näkyvä aktio kuvaa jotain toimintoa, mikä sovelluksessa voi tapahtua, yleensä käyttäjän toimien seurauksena. Näkymän päivityksestä voi seurata uusi toiminto, jonka dispatcher-komponentti ohjaa jälleen eteenpäin.

Dispatcher-komponentti on keskeisessä osassa Flux-sovellusta, sillä se hallinnoi kaikkea tiedon siirtymistä. Se on käytännössä rekisteri, johon kaikki store-komponentit rekisteröivät itsensä ja antavat takaisinkutsun (engl. callback), joka suoritetaan, kun jokin action-komponentti päätyy dispatcher-komponentin käsiteltäväksi. Store-komponenttien kutsut voidaan suorittaa tietyssä järjestyksessä, mikä on tärkeää vähänkin monimutkaisemmissa sovelluksissa. Ne myös odottavat, että kukin store-komponentti ehtii vuorollaan suorittaa haluamansa toiminnon loppuun asti, jolloin sovelluksen tila säilyy yhtenäisenä [12], [14]

Store-komponentti sisältää sovelluksen tilan ja toimintalogiikan. Käytännössä rooli on samantyyppinen kuin mallin MVC:ssä, mutta yhden kohteen sijasta ne kontrolloivat monen olion tilaa. Store rekisteröi siis itsensä dispatcherille takaisinkutsun kanssa, joka saa parametrina action-komponentin, kun dispatcher-komponentista suoritetaan kyseinen kutsu. Toiminnosta riippuen suoritetaan storen sisäisiä metodeita ja päivitetään sovelluksen tilaa. Päivityksen jälkeen näkymille lähtetetään tieto tilan muutoksesta, jotta nämä voivat päivittää itsensä. [14]

View-komponentti eli näkymä on vastuussa tietojen näyttämisestä käyttäjälle, kuten MVC:ssä, mutta korkealla näkymien hierarkiassa olevat näkymät ovat myös kontrollerin

roolissa. Ne kuuntelevat store-komponenttien tarjoamia päivityksiä ja välittävät niiltä saadun sovelluksen tilan kaikille hierarkiassa niiden alapuolella oleville näkymilleen. Kaikki näkymät päivitetään tämän uuden tilan perusteella, mikäli tarpeen ja tieto kulkee sovelluksessa vain yhteen suuntaan. Sovelluslogiikkaa saadaan lisäksi yksinkertaistettua välittämällä lapsinäkymille storen koko tila, jolloin jokainen näkymä voi käyttää siitä niitä osia, mitä se tarvitsee. Tällöin ei tarvitse ylläpitää esimerkiksi React-komponenttien staattisia osia eli propseja niin paljon. [14]

Action-komponentti on jokin sovelluksessa tietoa välittävä toiminto, jonka seurauksena dispatcher ohjaa tiedon eteenpäin store-komponenteille. Action-komponentteja voidaan luoda action creator-komponenttien avulla, jotka ovat apumetodeita action-komponenttien välittämiseen dispatcher-komponentille. Sovelluksen tila voi muuttua ainoastaan aktioiden kautta, joten ne ovat tärkeässä osassa sovelluksen toimintalogiikassa.

3. OFFLINE-TOIMINNALLISUUS

Web-sovellukset ovat verkkoyhteyden yli käytettäviä sovelluksia. Siitä huolimatta on tilanteita, joissa sovelluksen olisi hyvä toimia, kun verkkoyhteyttä ei ole saatavissa. Tämä saattaa johtua monestakin syystä, esimerkiksi kuuluvuus on huono tai yhteys on liian hidas. Tällöin sovelluksen käytettävyyden takaamiseksi sen olisi pystyttävä käsittelemään tietoja myös lokaalisti käyttäjän laitteella.

Yleisessä web-sovelluksen suoritusten kulussa siis selain ja palvelin keskustelevat keskenään eikä selaimeen juurikaan tallenneta tietoja, vaan ne ladataan aina palvelimelta tarpeen mukaan. Tämä tarkoittaa sitä, että verkkoyhteyden puuttuessa pitäisi selaimen pystyä toteuttamaan sekä sovelluslogiikkaa että tietokannan roolia. Siispä selaimen täytyy pystyä käsittelemään ja vastaamaan käyttäjän pyyntöihin. Lisäksi sen pitäisi pystyä sekä lukemaan että tallentamaan tietoja lokaalisti. Tässä luvussa käsitellään ensiksi offline-toiminnallisuutta ja sen periaatteita yleisellä tasolla. Myöhemmin esitellään matalamman tason toteutustapoja sovelluksen toimintaan ja tietojen tallentamiseen.

3.1 Offline first -periaate

Offline first on periaate, jonka mukaan sovellus tulisi suunnittella ensisijaisesti toimimaan offline-tilassa [16]. Tämän jälkeen verkkoyhteyden ollessa tarjolla voitaisiin sovelluksen tarjoamia ominaisuuksia mahdollisesti lisätä sen mukaan. Sen sijaan, että näytettäisiin virheilmoitus verkkoyhteyden puuttuessa, mikä todennäköisesti saisi käyttäjän lopettamaan sovelluksen käytön, voisi käyttäjä jatkaa sovelluksen käyttöä ja tehdä asioita lokaalisti, kunnes yhteys on jälleen saatavilla. [16], [17]

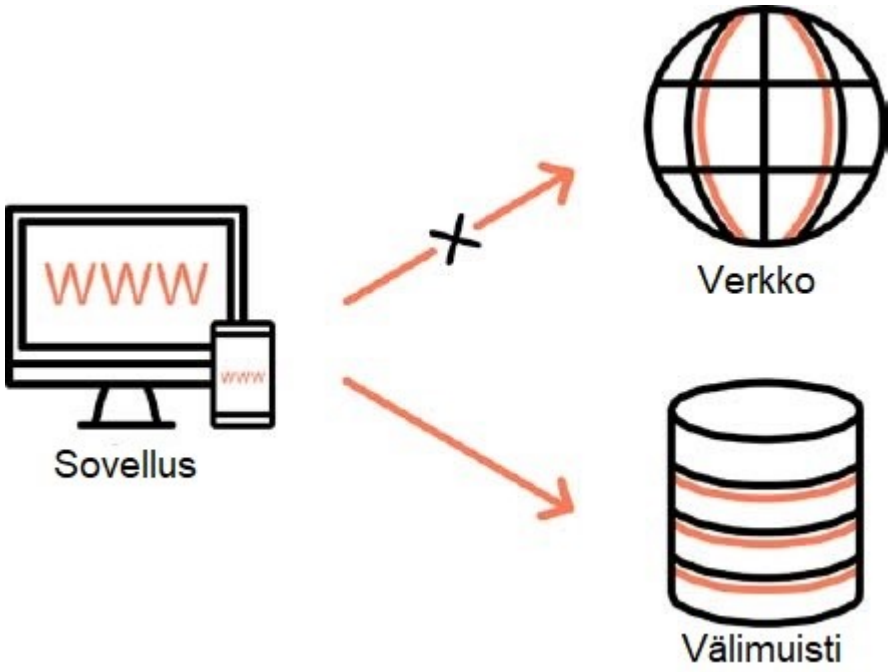
Feyerke esittää artikkelissaan [16] hyvän esimerkin offline-toiminnallisuutta tukevasta sovelluksesta sähköpostisovelluksen muodossa. Viestejä voidaan kirjoittaa ilman verkkoyhteyttä ja ne siirtyvät lähtevät kansioon odottamaan lähettämistä, kunnes yhteys verkkoon voidaan muodostaa. Tällöin ladataan myös uudet saapuneet viestit käyttäjän saataville lokaalisti. Tällaisessa sovelluksessa kaikki tiedot ovat käyttäjän saatavilla, vaikka hänellä ei olisi internet-yhteyttä. Vaikka tämä esimerkin sovellus ei olekaan web-sovellus, voidaan samoja periaatteita noudattaa niiden suunnittelussa ja toteutuksessa, jotta voidaan tarjota vastaavanlainen käyttökokemus käyttäjälle.

Web-sovellusten tapauksessa toki vaaditaan, että käyttäjä on navigoinut sivulle aikaisemmin, jotta tiedot on voitu tallentaa selaimeen. Toki myös sähköpostisovelluksen tie-

dot on synkronoitava käyttäjän laitteelle, ennen kuin ne ovat käytössä myös offline-tilassa. Monimutkaisempien ja paljon dataa sisältävien sovellusten tapauksessa on kuitenkin otettava huomioon, ettei käyttäjän koneelle voida ladata saman tien kaikkea mahdollista dataa offline-käyttöä varten. Tällöin täytyy tehdä kompromisseja sen suhteen, mitä tallennetaan kullekin käyttäjälle, mikä ei välttämättä ole yksiselitteisesti ratkaistavissa.

Progressiivinen web-sovellus

Progressiivinen web-sovellus (engl. Progressive Web Application) tarkoittaa sovellusta, joka toimii myös ilman verkkoyhteyttä. Se on käytännössä korkean tason periaate websovelluksien suunnitteluun. Sheppardin tarjoaman määritelmän mukaan sillä on seuraavat kolme ominaisuutta: nopeus, luotettavuus ja käyttäjän sitouttaminen [18]. Siispä sovelluksen lataaminen ei kestä kauaa ja se toimii myös ilman hyvää internet-yhteyttä. Lisäksi käyttäjälle voidaan näyttää ilmoituksia myös selaimen ollessa suljettuna, jolloin tuodaan asiat helposti käyttäjän ulottuville. PWA:n tulisi toimia niin vanhoilla kuin uusillakin laitteilla, mutta progressiivisuus tulee siitä, että uudemmilla laitteilla ja selaimilla voidaan tarjota enemmän ominaisuuksia, joita vanhemmat laitteet eivät välttämättä tue. Niinpä eri laitteiden käyttäjät eivät välttämättä näe samaa sisältöä.



Kuva 4. *Offline-tuki välimuistin avulla [18]*

Perusperiaatteena PWA:ssa on, että verkkoyhteyden puuttuessa käytetään välimuistia sisällön lataamiseen, kuten kuvassa 4 on esitetty. Sisältöä voidaan ladata muistista myös suorituskykysyistä, vaikka yhteys olisikin olemassa. Tarkoituksena olisi tarjota käyttäjälle

aina jotain näytettävää, jotta sovellus vaikuttaisi toimivan heikosta yhteydestä huolimatta. Jos tarvitsee ladata jotain verkkoyhteyden kautta, pyritään ensimmäisellä pyynnöllä lataamaan mahdollisimman vähän ja tämän jälkeen täydentämään sivun sisältöä AJAX-pyynnöillä. Progressiivinen web-sovellus toteuttaa siis tietynlaista suunnittellumallia, jonka tarkoituksena on tuottaa offline-toiminnallisuuden tarjoama web-sovellus. Käytettäviin työkaluihin siinä ei oteta kantaa, vaan se toimii ainoastaan ohjeina sovelluksen kehityksessä.

3.2 Korkean tason toteutus

Korkean tason vaihtoehtoja offline-toiminnallisuuden takaamiseksi on kolme erilaista mallia Andreun mukaan [19], jotka ovat seuraavat:

1. ladataan kaikki
2. välimuistin käyttö
3. valikointi.

Ensimmäinen vaihtoehto eli kaiken lataaminen on yksiselitteinen, mutta myös selkeitä heikkouksia omaava tapa. Tällöin ladataan ja tallennetaan lokaalisti heti käyttäjän kirjauduttua kaikki tiedot, mitä käyttäjä voi tarvita. Tämä saattaa kuitenkin olla todella hidasta etenkin, jos verkkoyhteys on huono. Myöskin datan hakeminen muistista saattaa olla hidasta, koska sitä on todennäköisesti paljon ja sovelluksen suorituskyky heikkenee datamäärän kasvaessa. Tämä heikentää käyttökokemusta heti alussa, kun käyttäjä joutuu odottelemaan lataamista ennen kuin pääsee tekemään sovelluksessa mitään. [19]

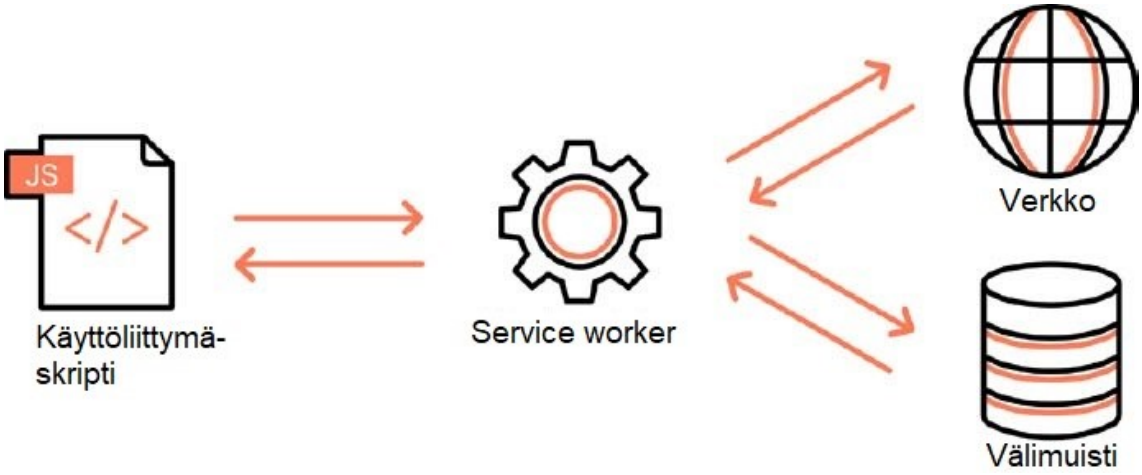
Toinen vaihtoehto on välimuistin käyttö, jossa tiedot haetaan palvelimelta ensimmäisellä kerralla, kun niitä pyydetään. Tämän jälkeen ne tallennetaan välimuistiin nopeuttamaan seuraavaa hakua, mikäli niitä tarvitaan uudestaan. Tämä lähestymistapa vaatii sen, että välimuistissa olevat tiedot pidetään ajan tasalla, kun mahdollista, esimerkiksi siten, että haettaessa tietoja muistista pyydetään niitä myös palvelimelta, ja päivitetään muistissa olevat tiedot, mikäli ne ovat muuttuneet. Tällaiset operaatiot on hyvä suorittaa taustalla asynkronisesti, jotta ne eivät hidasta sovelluksen käyttöliittymän käyttöä. [19]

Kolmantena vaihtoehtona voidaan valikoida, mitä halutaan tallentaa välimuistiin offline-käyttöä varten [19]. Käyttäjälle voidaan esimerkiksi tarjota mahdollisuus tallentaa haluamiaan kappaleita toistettavaksi tai joitakin alueita kartalla näytettäväksi myös offline-tilassa. Tällöin ei tarvitse tallentaa mitään ylimääräistä muistiin ja käyttäjä tietää, mihin tietoihin hänellä on offline-tilassa pääsy. Eri periaatteita voidaan toki soveltaa sopiviksi haluttuihin käyttötapauksiin offline-toiminnallissuutta toteutettaessa eikä tarvitse pysy-

tellä vain yhdessä toimintatavassa. Tämä on myös todennäköisesti kannattava tapa toimia, sillä sovelluksella voi olla hyvin monen tyyppisiä resursseja, joita offline-tilassa saatetaan tarvita.

3.3 Service worker

Service Worker API tarjoaa rajapinnan web-sovelluksen ja palvelimen välille. Sen avulla voidaan esimerkiksi ladata tallennettuja resursseja muistista ilman verkkoyhteyttä. Service Workereita ajetaan eri säikeessä kuin käyttäjän antamia käyttöliittymäkomentoja [18]. Tällöin se ei hidasta käyttöliittymän toimintaa, vaan toimii ikään kuin taustalla. Service workerit eivät itsessään tarjoa rajapintaa tietojen tallentamiseen, vaan ne tarvitsevat jonkin tiedon varastointitavan tuekseen. Service workerit toimivat asynkronisesti, jolloin muun koodin suoritus ei joudu odottamaan operaatioiden valmistumista.



Kuva 5. *Service workerien toimintaperiaate [18]*

Service workerit alustetaan käyttöliittymäskriptissä, jonka jälkeen niiden suoritus tapahtuu omassa silmukassaan. Ne lataavat resursseja sekä välimuistista että internetin välityksellä riippuen niiden saatavuudesta. Kuvassa 5 on havainnollistettu tätä toimintaperiaatetta. Kun selaimesta olisi lähdössä pyyntö palvelimelle, service workerille rekisteröity kuuntelija kaappaa tämän pyynnön. Tämän jälkeen voidaan hakea vastaus välimuistista, jos se on olemassa tai antaa alkuperäisen pyynnön mennä palvelimelle. Tässä aliluvussa tarkastellaan, kuinka service workereita voidaan hyödyntää niin resurssien lataamiseen kuin niiden tarjoiluun käyttäjälle.

Pyyntöjen tallentamiseen service workerissä voidaan käyttää *Cache APIa*, jonne voidaan tallentaa verkkopyynnöistä ja vastauksista koostuvia pareja. Cache-rajapinta luotiin service workereitä silmällä pitäen, jotta niiden avulla voitaisiin tallentaa väli muistiin verkosta saatuja vastauksia. Siinä mielessä se soveltuu erityisen hyvin service workereiden kanssa käytettäväksi. Rajapintaa voi toki käyttää myös service workereiden ulkopuolella

yleisenä vastausten tallennusmekanismina, sillä se on saatavilla selaimessa globaalissa *window*-muuttujassa. [20]

3.3.1 Resurssien lataaminen verkosta

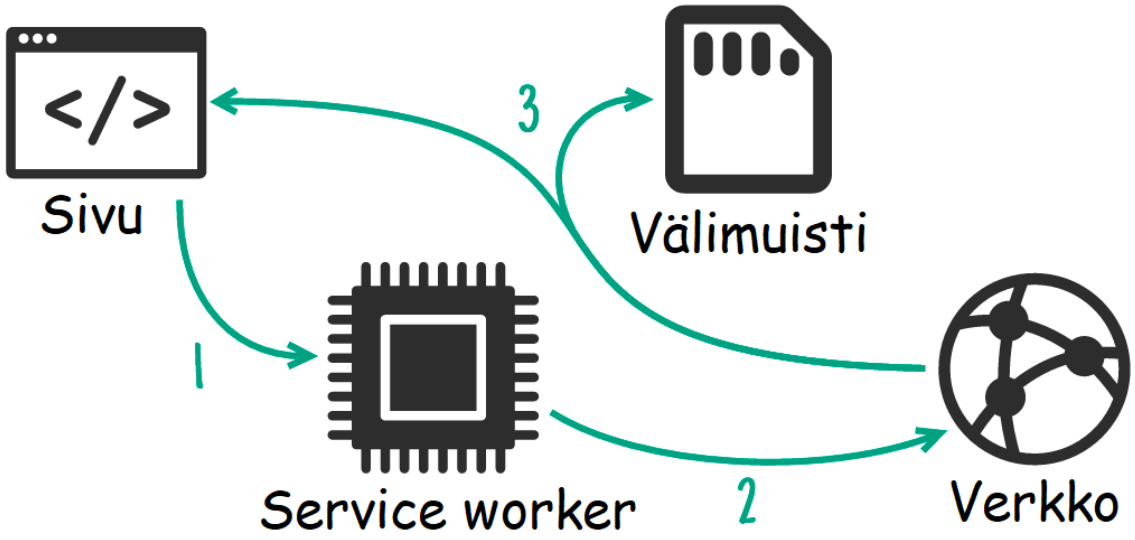
Aliluvussa 3.2 käsiteltiin korkealla tasolla erilaisia vaihtoehtoja resurssien lataamisessa välimuistiin sovelluksen offline-käyttöä varten. Nyt tarkastellaan hieman tarkemmin service workereita ja niiden hyödyntämistä erilaisten resurssien kanssa sekä sitä, milloin resurssien lataaminen olisi syytä tehdä. Service workerien osalta käsitellään seuraavia Archibaldin esittelemiä [21] tallennusvaihtoehtoja:

- service workerin asennus
- service workerin aktivointi
- käyttäjän syöte
- verkosta saatu vastaus.

Kun sivu ladataan ensimmäisen kerran, täytyy service worker alustaa. Tässä asennusvaiheessa on hyvä tallentaa kaikki sovelluksen staattinen sisältö välimuistiin. Staattisella sisällöllä tarkoitetaan tässä tapauksessa CSS-tiedostoja, kuvia, fontteja, JavaScript-tiedostoja ja template-tiedostoja. Ilman näitä resursseja sovellusta ei voi käyttää, joten ne on ladattava heti aluksi [21]. Ne kuitenkin muuttuvat harvoin, joten niitä ei tarvitse istunnon aikana ladata uudestaan. Kaikkia näitä resursseja ei välttämättä tarvita heti, joten on mahdollista ladata ensin vain välttämättömät tiedostot ja yrittää ladata muita myöhemmin uudelleen.

Asennuksen onnistuttua ja mahdollisen vanhan service workerin saatua pyynnöt käsiteltyä, tapahtuu uuden workerin aktivointi. Tällöin voidaan poistaa välimuistista tarpeettomia vanhan service workerin tallentamia tietoja ja päivittää esimerkiksi selaimessa käytetyn tietokannan tietomalli. Tässä vaiheessa muut tapahtumat, kuten resurssien haku (engl. fetch), joutuvat jonoon, joten aktivoinnissa kannattaa tehdä mahdollisimman vähän asioita, jottei hidasteta sivun latautumista. [21]

Aikaisemmin käsitellyissä korkean tason periaatteissa käytiin jo läpi valikoiva sisällön tallentaminen, jossa käyttäjä voi valita, mitä resursseja haluaa käyttöönsä offline-tilassa. Tätä kannattaa Archibaldin mukaan [21] hyödyntää, kun koko sivustoa ei pystytä offline-tilaan tallentamaan. Esimerkiksi kun sisältöä on niin paljon, että on parempi antaa käyttäjän valita. Tätä voidaan hyödyntää myös ilman service workereita, haetaan vain käyttäjän pyytämä resurssi verkosta ja tallennetaan se välimuistiin.



Kuva 6. *Tallentaminen välimuistiin, kun saadaan vastaus verkosta [21].*

Aikaisemmin käsiteltiin myös välimuistin käyttöä siten, että tallennetaan resursseja sitä mukaa, kun käyttäjä niitä pyytää. Tätä vastaa Archibaldin esittelemistä periaatteista [21] tallennus muistiin, kun saadaan vastaus verkosta. Kuvassa 6 on esitetty tämän prosessin kulku. Service worker hakee siis ensin vastauksen verkosta ja tallentaa tämän jälkeen vastauksen muistiin sekä palauttaa vastauksen sivulle. Tätä kannattaa hänen mukaansa hyödyntää usein päivittyvien resurssien kohdalla, kuten käyttäjälle saapuneet viestit tai artikkelin sisältö. Lisäksi tätä voidaan käyttää resursseissa, jotka eivät ole niin välttämättömiä, kuten profiilikuvat. Tähän liittyy myös toinen tapa, jossa käytetään muistissa olevaa versiota esimerkiksi profiilikuvasta, mutta haetaan verkosta silti tieto uusin tieto, joka tallennetaan seuraavaa käyttöä varten muistiin. [21]

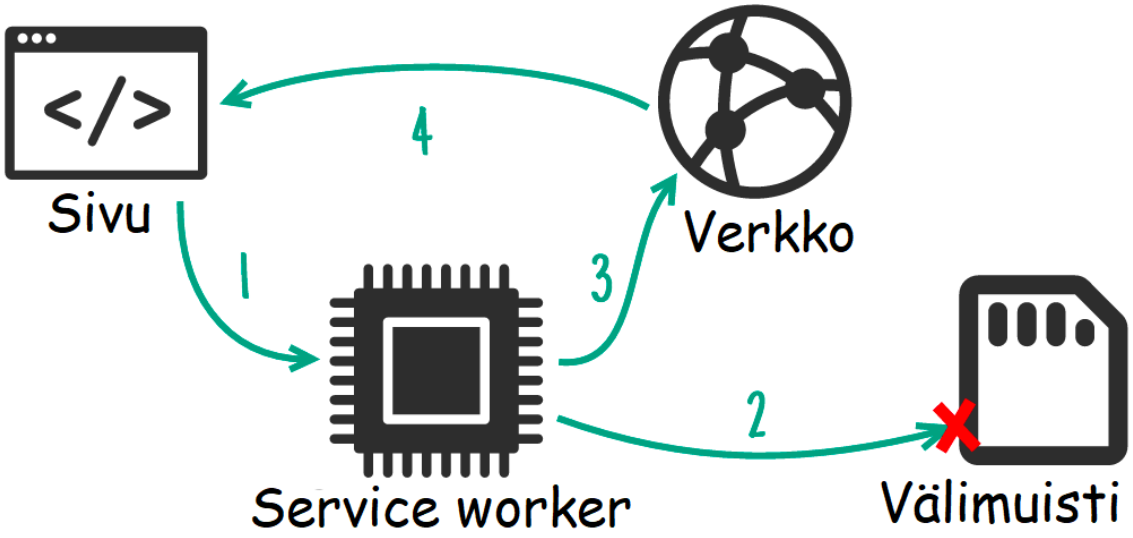
Näiden lisäksi Archibald mainitsee [21] kaksi taustalla suoritettavaa esimerkkiä, jolloin tietoja voidaan hakea välimuistiin. Kun käyttäjälle lähetetään ilmoitus esimerkiksi selaimen ollessa suljettuna, olisi hyvä ladata siihen liittyvä resurssi muistiin, jotta käyttäjän hakiessa ilmoitukseen liittyvää resurssia, se olisi aina saatavilla. Tätä voidaan hyödyntää epäsäännöllisesti päivittyvissä tiedoissa, kuten kalenterin muutoksissa tai sähköpostiviesteissä. Lisäksi taustalla voidaan ajastetusti synkronoida sellaisia tietoja, jotka päivittyvät liian usein, jotta niistä kannattaisi lähettää käyttäjälle ilmoituksia. Esimerkkinä tällaisesta mainitaan pelin ennätystuloslista [21].

3.3.2 Resurssien käyttö

Resurssien lataamiseen ja tallentamiseen liittyy tietenkin läheisesti myös niiden tarjoaminen käyttäjälle niitä pyydettäessä. Archibald esittelee artikkelissaan [21] myös tähän

monenlaisia ratkaisuvaihtoehtoja ja ehdotuksia, milloin tulisi käyttää mitäkin tapaa resurssien hakemiseen käyttäjälle. Tässä käsiteltävät tavat ovat:

- vain välimuisti
- välimuisti, verkko varalla
- välimuisti, jonka jälkeen verkko
- vain verkko
- verkko, välimuisti varalla
- välimuistin ja verkon kilpailu.



Kuva 7. *Välimuisti pyynnön ensisijaisena kohteena, jonka jälkeen verkko varalla – lähteestä mukailtuna. [21]*

Pelkän välimuistin käyttö sopii hyvin staattisille resursseille, jotka tallennettiin heti service workerin alustusvaiheessa [21]. Niinpä nämä voidaan hakea muistista aina seuraavilla pyynnöillä olettaen, että alustus onnistui ensimmäisessä vaiheessa. Verkkoa voi toki käyttää varalla aina välimuistista haettaessa, mikäli yhteys on mahdollinen. Muutoin verkkoa tulisi käyttää välimuistin varalla esimerkiksi offline first -periaatteita noudattavassa sovelluksessa [21]. Tällöin voidaan aina ensiksi tarkistaa, löytyykö resurssia välimuistista, jonka jälkeen haetaan se verkosta, jos sitä ei löytynyt. Tämä prosessi on esitetty kuvassa 7. Tämä takaa sen, että verkkoa käytetään resursseihin, joita ei voi tallentaa välimuistiin, kuten muut kuin lukuoperaatiot. Lisäksi kolmas tapa, jossa haetaan välimuistista resurssia, on hakea se ensin käyttäjälle, jonka jälkeen haetaan verkosta päivitetty tieto, joka voidaan tämän jälkeen myös päivittää käyttäjän näkyville. Tämä on hyödyllistä, kun tieto päivittyy usein esimerkiksi sosiaalisen median tapauksissa. Tällöin käyttäjä näytetään heti jotain, jolloin ei tarvitse odottaa verkon vastausta. [21]

Verkon käyttö ensisijaisena välineenä on joissain tapauksissa välttämätöntä, koska vastaavuutta ei voida tarjota offline-tilassa. Näitä ovat tietoja muokkaavat operaatiot, joita ei toki tarvitse käsitellä erillistapauksena, jos sovelluksessa on käytetty oletuksena verkkoa välimuistin varalla, kuten aiemmin mainittu. Usein päivittyvien resurssien yhteydessä verkon käyttö ensijaisesti välimuistin ollessa varalla on paras vaihtoehto. Tämä tarkoittaa, että offline-tilassa olevat käyttäjät joutuvat tyytymään välimuistissa olevaan vanhaan sisältöön ja online-käyttäjät näkevät tuoreimmat päivitykset. Kuitenkin, verkkoyhteyden ollessa hidas voi olla parempi hakea ensin muistista käyttäjälle jotain nähtävää ja sen jälkeen yrittää päivittää tiedot verkosta. [21]

Joissakin suorituskykyä vaativissa tilanteissa voi olla paras vaihtoehto hakea resurssia sekä välimuistista ja verkosta, ja tarjota käyttäjälle nopein saatu vastaus. Tämä voi olla hyödyllistä, jos haettava resurssi on pieni, ja tiedon levyltä lukeminen on jostain syystä hidasta esimerkiksi käyttäjän laitteistosta johtuen. Näiden vaihtoehtojen lisäksi voi olla hyvä tarjota käyttäjälle jokin geneerinen ilmoitus, mikäli resurssia ei ole mahdollista hakea sillä hetkellä. [21]

3.4 Tietojen tallennus

Sovelluksen staattisten resurssien lisäksi sovelluksen dataa tulisi pystyä tallentamaan selaimessa johonkin, josta se olisi helposti luettavissa ja muokattavassa offline-käytön mahdollistamiseksi. Tähän on eri selaimien tarjoamasta tuesta riippuen tarjolla useampia vaihtoehtoja, joista valitessa ratkaisevina tekijöinä ovat muun muassa tallennettavan datan määrä ja rakenne. Yksinkertaisten ja pienten tietorakenteiden tallentaminen monimutkaisimpia tallennusmenetelmiä käyttäen ei välttämättä ole paras ratkaisu. Seuraavaksi esitellään lyhyesti erilaisia vaihtoehtoja datan tallentamiseen selaimessa.

Web Storage API tarjoaa yhden mahdollisuuden lokaaliin tietojen tallentamiseen selainympäristössä. Se sisältää kaksi vaihtoehtoista tapaa riippuen siitä, kuinka tietojen halutaan säilyvän. Jos tietoja tarvitaan vain nykyisen istunnon ajaksi, sessionStorage tarjoaa siihen mahdollisuuden. Tiedot säilyvät selaimessa niin kauan, kun sitä ei suljeta. Toinen mahdollisuus on localStorage, joka säilyttää tiedot myös selaimen sulkemisen jälkeen. Lisäksi tallennetut tiedot näkyvät kaikille välilehdille ja ikkunoille vain yhden sijasta toisin kuin sessionStoragen kohdalla [22]. [23]

Toisin kuin evästeiden kohdalla, nämä selaimessa tallennetut tiedot eivät välity jokaisella pyynnöllä palvelimelle ja takaisin. Lisäksi tallennetun datan koko voi olla huomattavasti suurempi kuin evästeissä. Nykyisin käytetyistä selaimista kaikki tukevat Web Storagea,

joten se on hyvä valinta, kun halutaan taata toimivuus useimmissa selaimissa. Sen heikkoutena puolestaan voidaan pitää sitä, että kaikki tiedot on tallennettava merkkijonoina, jolloin monimutkaisempien taulukoiden ja olioiden tallentaminen ei onnistu suoraan. Tämä ongelma voidaan kiertää JSONin avulla. Ennen tallentamista muunnetaan olio merkkijonoksi ja lukemisen jälkeen jäsennetään merkkijonosta jälleen olio. [22]

Välimuistiin voidaan tallentaa staattisia tiedostoja, mutta tietokannan tiedotkin pitäisi tallentaa johonkin selaimessa, jos niihin halutaan päästä käsiksi offline-tilassa. Tähän on tarjolla useampia vaihtoehtoja vaihtelevalla selaintuella [18]. IndexedDB on asynkroninen tietokanta, joka tarjoaa mahdollisuuden suurempien tietomäärien tallentamiseen selaimessa. Rajapinta käyttää indeksejä hakujen suorituskyvyn parantamiseksi. Sen tuki Internet Explorerille ja Microsoft Edgelle on vain osittainen [24], joten se ei välttämättä tule kyseeseen joka tilanteessa vaihtoehdoksi. Monimutkaisten tietorakenteiden kohdalla se on kuitenkin parempi ratkaisu kuin Web Storage.

LocalForage yksinkertaistaa selaimessa käytettävän tietokannan käyttöä esimerkiksi IndexedDB:hen verrattuna [18], [25]. LocalForage tarjoaa rajapinnan, jolla voidaan käyttää useampaa tietojen tallennustapaa selaimen tuesta riippuen. Jos selain ei tue IndexedDB:tä, käytetään localStoragea varavaihtoehtona.

3.5 Tietojen synkronointi

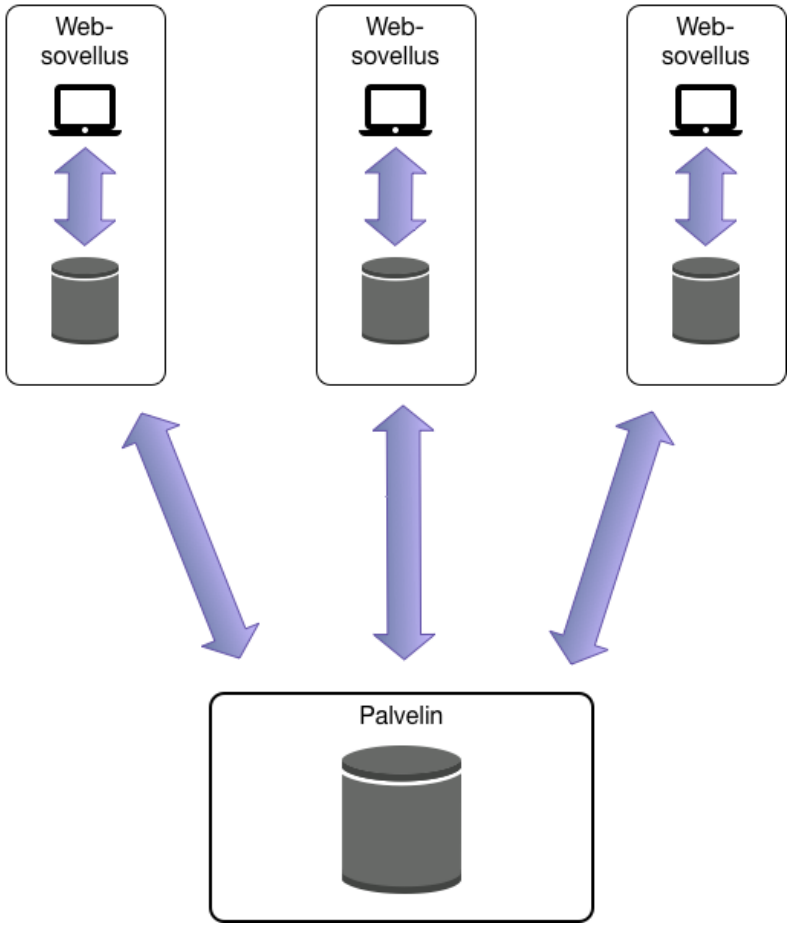
Tietojen lokaalin tallentamisen lisäksi tarvitaan jokin keino niiden synkronointiin takaisin palvelimelle. Offline first -arkkitehtuurin mukaiset sovellukset käyttävät ensisijaisesti lokaalia dataa ja tiedot on tallennettu omaan selaimessa sijaitsevaan tietokantaan. Tätä tietokantaa pyritään päivittämään aina verkkoyhteyden ollessa saatavilla synkronoimalla muutokset palvelimen tietokannan kanssa [26]. Kuvassa 8 on esitetty tällaisen arkkitehtuurin rakenne.

Huculakin mukaan [26] selaimessa ja palvelimella sijaitsevien tietokantojen kommunikaatioon on olemassa seuraavanlaisia strategioita:

- push-ilmoitukset
- tietokantojen synkronointi
- etukäteen lataaminen.

Älypuhelimista tutut push-ilmoitukset voivat olla hyvä ratkaisu tietojen synkronointiin, jos palvelin on ensisijainen datan lähde ja selain toimii lähinnä datan kuluttajana. Huculak mainitsee esimerkkinä tällaisesta uutissovellukset. Tällöin palvelimen tietojen muuttuessa lähetetään asiakkaalle ilmoitus, jonka saatuaan asiakas päivittää tietokantansa. Mikäli päivitettävää tietoa on vähän, voidaan kaikki data lähettää ilmoituksen sisällössä.

Tämän tyyppistä tapaa voidaan käyttää esimerkiksi Twitter-päivityksissä tai säätiedoissa. Jos dataa on enemmän, selain saa ilmoituksen, että dataa olisi saatavilla ja voi itse päättää, milloin tiedot haetaan palvelimelta. Tätä tapaa sopii puolestaan käytettäväksi esimerkiksi sähköpostisovellusten tai mediatoistinten kanssa, kun dataa tarvitaan kerralla paljon. [26]



Kuva 8. *Offline first-arkkitehtuuri, jossa jokaisella asiakkaalla on oma lokaali tietokantansa pohjautuen lähteeseen. [27]*

Tietokantojen keskinäiseen synkronointiin on puolestaan pääasiassa kaksi erilaista lähestymistapaa. Voidaan joko käyttää olemassa olevia tietokantaratkaisuja, jotka synkronoivat lokaalin ja palvelimen tietokannan automaattisesti keskenään tai synkronointi voidaan toteuttaa itse, jos valmiit teknologiat eivät tule kyseeseen. Teixeira mainitsee [27] CouchDB:n ja PouchDB:n, jotka tarjoavat replikointiprotokollan tietokantojen synkroinitiin selaimen ja palvelimen välillä. Itse toteutettaessa tulisi Huculakin mukaan ottaa huomioon, milloin synkronointi suoritetaan, ja kuinka vältetään konflikteja tietokantojen välillä. Lisäksi pitää päättää, kuinka konfliktit selvitetään ja mitä ylimääräistä informaatiota täytyy tietokantoihin tallentaa synkronointeja varten. [26]

Tietojen lataamiseen palvelimelta etukäteen on myös kaksi erilaista tapaa. Ensinnäkin voidaan seurata käyttäjän toimia ja tästä päätellä, mitä dataa saatetaan tarvita lähitulevaisuudessa. Tämän perusteella voidaan esimerkiksi ladata valmiiksi toistettavia kappaleita, tai kartalla äskettäin haettujen alueiden lähialueita. Toinen tapa on ladata paljon dataa valmiiksi silloin, kun yhteys on nopea esimerkiksi, kun mobiililaitteen käyttäjä on yhteydessä WiFi-verkkoon. Tällöin ei tarvitse ladata matkapuhelinverkkojen välityksellä, mikäli käsitellään suuria määriä dataa. [26]

3.5.1 Synkronointitekniikat

Datan synkronointi voi olla yksi- tai kaksisuuntaista. Esimerkiksi web-sovellusten kaltaisessa asiakas-palvelin-mallissa datan muutokset voivat tapahtua vain palvelimella, jolloin synkronoidessa asiakkaan tarvitsee vain ladata tietoja. Joissain sovelluksissa taas muutokset tapahtuvat asiakkaan puolella ja ne ainoastaan ladataan palvelimelle, mutta palvelimen data ei muutu muuten. Näissä tapauksissa kyse on yksisuuntaisesta synkronoinnista. Kaksisuuntaisessa synkronoinnissa puolestaan tehdään molempia edellä mainituista [28]. Palvelimen data voi muuttua sillä aikaa, kun asiakas tekee lokaalisti omia muutoksiaan.

Muutosten jälkeen tiedot täytyy synkronoida yhteneviksi, mikäli mahdollista. Faiz ja Shanker [28] esittelevät artikkelissaan seuraavat tekniikat datan synkronointia varten:

- kokonaisvaltainen synkronointi
- tilapohjainen synkronointi
- aikaleimaan perustuva synkronointi
- matemaattinen synkronointi
- viestin tiivisteeseen perustuvat synkronointi
- lokipohjainen synkronointi.

Kokonaisvaltaisessa synkronoinnissa toinen osapuoli lähettää kaiken sisältämänsä datan toiselle lokaalia vertailua varten. Tämä on etenkin suurten datamäärien kohdalla tehotonta ja hidasta, mutta helppo toteuttaa ja varmistua siitä, että kaikki muutokset siirtyvät. Tilapohjaisessa menetelmässä asiakas käyttää lippua (engl. *flag*) ilmaisemaan, onko jokin alkio luotu, päivitetty tai poistettu. Synkronoidessa voidaan lähettää ainoastaan muutokset, joille lippu on asetettu. Tämä on tehokkaampaa kuin kaiken synkronoimisen kohdalla, mutta jos asiakkaita on useampia, täytyy tietää, kenen kanssa on synkronoitu mitäkin, jolloin logiikka monimutkaistuu. Aikaleimaan perustuvassa menetelmässä pidetään kirjaa, milloin tietoa viimeksi muokattiin, ja milloin kunkin asiakkaan tiedot edellisen kerran synkronoitiin. Tämä helpottaa tilapohjaisen menetelmän ongelmia, kun ei tarvitse pitää yllä tietoa siitä, kenen kanssa tiedot on synkronoitu. [28]

Matemaattisessa synkronoinnissa käytetään tiedon sisältämiä matemaattisia ominaisuuksia, esimerkiksi polynomeja kuvaamaan dataa ja sen muutoksia. Viestin tiivisteeseen perustuva synkronointi on eräs matemaattisen synkronoinnin muoto. Siinä tiivisteen perusteella lasketaan muuttunut data, joka täytyy synkronoida. Viimeisenä menetelmänä mainittiin lokipohjainen menetelmä, jossa jokainen muutos suoritetaan transaktiona järjestelmään ja tallennetaan lokiin. Lokit voidaan synkronoida muiden asiakkaiden kanssa ja tämän avulla myös tiedot synkronoituvat, kun kaikki operaatiot toistetaan vastaanottavan asiakkaan päässä. Tämänkaltainen ratkaisu on yleinen esimerkiksi versionhallinnassa. Näitä tekniikoita voidaan myös yhdistää optimaalisten tiedon synkronointitapojen saavuttamiseksi. [28]

Useamman asiakkaan tehdessä muutoksia offline-tilassa samaan data-alkioon voi syntyä konflikteja. Feyerken mukaan käyttäjälle olisi hyvä tarjota konfliktien ratkaisuun jokin työkalu, jossa on mahdollisuus valita, mitä versiota tiedoista käytetään [16]. Konfliktilanteissa esimerkiksi PouchDB ja CouchDB pitävät kaiken datan tallessa, jolloin ohjelmoijalle on jätetty vapaus valita haluamansa tapa ratkaista tilanne [17]. Faiz *et al.* [28] luettelevat seuraavia kirjallisuudessa esitettyjä konfliktinratkaisuun tarkoitettuja menetelmiä:

- lähettäjä voittaa
- vastaanottaja voittaa
- asiakas voittaa
- palvelin voittaa
- uusin tieto voittaa
- kopiointi, molemmat säilytetään.

Näistä menetelmistä voidaan valita järjestelmän käsittelemästä datasta sekä kyseessä olevan tiedon tyypistä riippuen sopiva menetelmä tai sitten käyttää aina samaa metodia konfliktien ratkaisuun.

4. TOTEUTUS

Tässä luvussa tarkastellaan tämän diplomityön konstruktiiviseen osaan liittyviä tekijöitä eli ympäristöä, johon toteutus on tehty ja toteutuksessa huomioitavia seikkoja offline-toiminnallisuuden ja tietojen synkronoinnin suhteen. Lisäksi käsitellään offline-toiminnallisuuden toteutuksessa mukana olleita tekniikoita.

4.1 Ympäristö

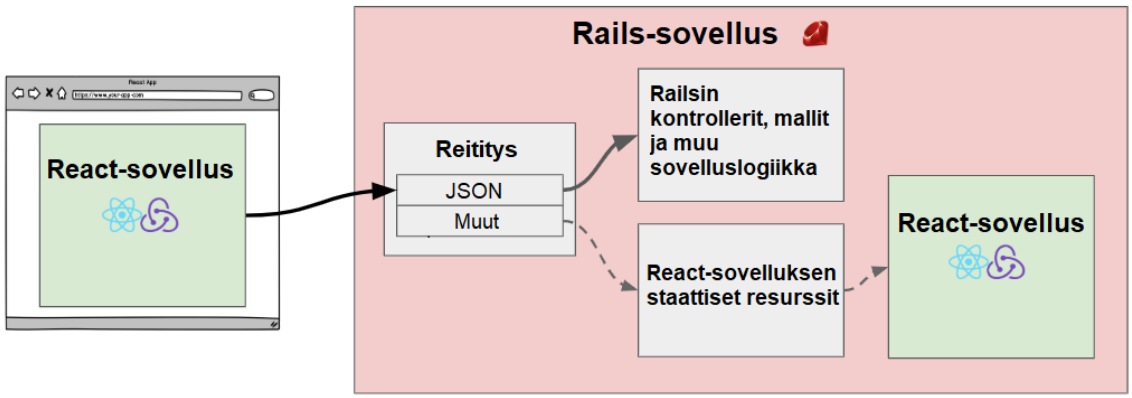
Tämän diplomityön toteutus on tehty Adalia Oy:lle osana asiakasprojektia, jossa kohteena on maatalousyrittäjien työterveyshuoltoon suunnattu web-sovellus. Asiakkaalle toteutettavan sovelluksen yhtenä päätoiminnallisuutena ovat maatiloille kohdistuvat tilakäynnit. Tilakäynnille voi osallistua useampia henkilöitä, joita voivat olla esimerkiksi työterveyshuollon työntekijät ja maatalouden asiantuntijat. Näiden henkilöiden on tarkoitus käyttää järjestelmää maatilalla kiertelyn ohessa mahdollisten havaintojensa kirjaamiseen, esimerkiksi mahdollisten terveydenvaarojen osalta. Tilakäynnin havaintojen perusteella voidaan muodostaa raportti järjestelmään. Tilakäyntien avulla valvotaan vakuutuksen ottaneiden maatalousyrittäjien työolosuhteita. Asiakkaana projektissa on Mela eli Maatalousyrittäjien eläkelaitos, joka on erilaisia työterveyshuoltoon liittyviä palveluita, kuten vakuutuksia ja apurahoja, tarjoava organisaatio.

4.1.1 Järjestelmän arkkitehtuuri

Asiakasprojektin järjestelmä on Railsilla toteutettu web-sovellus, jonka käyttöliittymä on toteutettu Reactin avulla SPA-periaatteiden mukaisesti. Niinpä Rails-sovelluksen backend toimii ensimmäisen sivulatauksen jälkeen käytännössä ainoastaan rajapintana, joka palauttaa frontendille tietoja JSON-muodossa. Ensimmäisellä sivulatauksella palvelimelta haetaan sovelluksen tarvitsemat staattiset resurssit eli sovelluksen pohjana toimiva HTML-dokumentti sekä JS- ja CSS-tiedostot. Näiden pyyntöjen kulku on esitetty kuvassa 9 yhtenäisillä nuolilla JSON-pyyntöjen osalta ja ensimmäisellä latauksella tehtävä staattisten resurssien haku puolestaan katkoviivoin. React-sovelluksen hakeva pyyntö tulee luonnollisesti selaimelta eikä React-sovellukselta, jota ei ole vielä ladattu.

Järjestelmä on siis MVC-arkkitehtuurimallin mukainen sovellus, jossa näkymän roolista vastaa selaimessa toimiva React-sovellus. Railsin kontrollerit käsittelevät käyttöliittymäsovelluksen tekemät JSON-pyynnöt ja hakevat tietoa malleilta MVC-periaatteiden mukaisesti. Jokainen malli vastaa yhtä tietokannan taulua, kuten Railsissa on tapana ja

lisäksi jokaista mallia kohden on yleensä yksi kontrolleri, joka vastaa kyseiseen malliin liittyvistä kyselyistä ja tietojen päivittämisestä.



Kuva 9. *React-sovelluksen toiminta osana järjestelmää pohjautuen lähteessä esitettyyn malliin [29].*

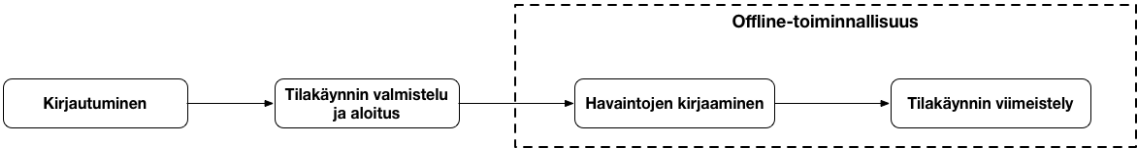
Käyttöliittymän toteuttava React-sovellus voidaan siis mieltää omaksi selaimessa toimivaksi sovelluksekseen, joka käyttää rajapintanaan Rails-sovelluksen tarjoamia tietoja. Tällöin Rails-sovellus ei ole tietoinen käyttöliittymäsovelluksen toiminnasta. Etenkin, kun offline-käytössä tilakäyntien yhteydessä päätelaitteen ja palvelimella sijaitseva data eivät täsmää. Offline-käytössä sovelluksen tietoja tallennettaan lokaalisti ja tällöin laitteelta löytyvät tiedot kuvaavat sovelluksen tilaa. Online-tilassa nämä tiedot voidaan synkronoida, jonka jälkeen selainsovellus ja palvelin ovat samassa tilassa.

4.1.2 Offline-toiminnallisuus osana järjestelmää

Tämän diplomityön yhteydessä toteutettu offline-toiminnallisuus on sovelluksen tilakäyntien liittyvien tietojen tallentaminen käyttäjän laitteelle, kun käyttäjä aloittaa tilakäynnin online-tilassa. Tämän jälkeen havaintojen kirjaaminen on mahdollista offline-tilassa. Kuvassa 10 on esitetty offline-toiminnallisuuden osuus tilakäyntien suhteen järjestelmässä. Tästä nähdään myös, että tilakäyntien viimeistely voidaan suorittaa offline-tilassa. Mikäli käynnin kohteena olevan maatilan maatalousyrittäjä ei ole antanut lupaa mahdollisten terveydenvaarojen kirjaamiseen järjestelmään, ei terveydenvaaroja synkronoida palvelimelle ollenkaan, vaan ne tallennetaan ainoastaan paikallisesti. Muut tilakäyntiin liittyvät tiedot synkronoidaan palvelimelle, kun verkkoyhteys on käytettävissä.

Offline-toiminnallisuus on siis tarkoitus tarjota vain järjestelmässä tehtävien tilakäyntien suhteen, jolloin muuta järjestelmää ei ole mahdollista käyttää offline-tilassa. Tilakäynneillä tehtyjä havaintoja, esimerkiksi mahdollisia terveysuhkia, tulisi pystyä kirjaamaan järjestelmään. Järjestelmästä tulisi löytyä tilan tiedot ja sieltä löytyvät tarkastettavat koh-

teet. Tilakäynnit ovat sovelluksen toiminnallisuuden kannalta pääosassa, ja siinä mielessä toimivuus myös offline-tilassa on kriittinen osa-alue. Tilakäynnit kohdistuvat siis maatiloille, joilla liikuttaessa ollaan todennäköisesti mobiiliverkon varassa. Sen toimivuuteen syrjäisillä maaseudun alueilla ei voida luottaa, joten offline-tuen avulla voidaan varmistaa, että tiedot ovat tallennettuna vähintään yhdelle laittelle, mikäli niitä ei saada saman tien synkronoitua palvelimelle.



Kuva 10. *Offline-toiminnallisuuden osuus tilakäyntien suorittamisesta järjestelmässä.*

Koska järjestelmä on vasta toteutusvaiheessa, ei tilakäyntien toiminnallisuutta ollut toteutettu millään tavalla ennen tämän työn aloittamista. Siispä koko ominaisuuden toiminnallisuuden suunnittelussa ja toteutuksessa voitiin huomioida mahdolliset offline-toiminnallisuuden aiheuttamat haasteet. Toteutus voitiinkin tehdä offline-toiminnallisuuden ehdoilla huomioiden olemassa olevan toiminnallisuuden asettamat rajoitukset, kuten käytettävät tekniikat. Tässä työssä ei varsinainen tilakäynteihin liittyvä toiminnallisuus ole tarkastelun kohteena, vaan tekniset ratkaisut offline-tilassa tapahtuvan tietojen tallentamisen sekä palvelimelle synkronoimisen takana.

Tietojen tallennuksen mahdollistamiseksi on otettava huomioon mahdolliset tilanteet, joissa sovellus joudutaan lataamaan uudelleen, jolloin pelkästään tilakäyntien datan tallentaminen paikallisesti ei mahdollista kirjausten tekemisen jatkamista offline-tilassa. Tästä syystä sovelluksessa täytyy toteuttaa myös staattisten resurssien tallentaminen, jotta selaimessa voidaan ladata sovellus uudestaan myös ilman verkkoyhteyttä.

Tilakäynteihin liittyvän datan tallentaminen tehdään siinä vaiheessa, kun käyttäjä aloittaa tilakäynnin ollessaan online-tilassa. Tämä johtuu siitä, että vaatimusmäärittelyssä on mainittu, että tilakäyntiä on pystyttävä jatkamaan offline-tilassa. Tämä siis sen varalta, että yhteys katkeaa kesken käynnin. Toteutuksessa on kuitenkin hyvä huomioida myös se mahdollisuus, että tulevaisuudessa halutaan automaattisesti ladata esimerkiksi saman päivän tilakäynnit käyttäjän laitteelle tämän kirjautuessa. Käyntejä voi olla useita päivässä, joten käyttäjän vastuulla olisi tuolloin vain kirjautuminen järjestelmään, jonka jälkeen voidaan tilakäyntien tiedot synkronoida selaimeen. Etukäteen tallentamisen lisäksi käynnillä tehtävät syötteet tulisi tallentaa lokaalisti, jottei muutoksia menetetä käyttäjien sulkiessa selaimen.

Vaikka tässä työssä pääpainopisteenä onkin tietojen tallentaminen ja synkronointi, täytyy myös tietoturva ottaa huomioon. Offline-käytössä yksi tähän liittyvä asia on käyttäjän tunnistautuminen. Kun sovelluksen resurssit ja data on ladattu käyttäjän laitteelle, ei offline-käytössä voida käyttäjää pyytää kirjautumaan istunnon vanhentuessa. Tietenkin ennen palvelimelle synkronointia täytyy käyttäjän kirjautua uudestaan, mikäli istunto ei ole voimassa. Tässä välillä on kuitenkin mahdollista, että tietoja on käsitellyt myös joku muu. Tässä voitaisiin vielä lisäksi pyytää tunnistautumisen jälkeen käyttäjää vahvistamaan tiedot ennen niiden synkronointia palvelimelle. Tällöin käyttäjä olisi vastuussa siitä, että tiedot pitävät paikkansa.

4.1.3 Tietojen synkronointi palvelimen kanssa

Tietojen synkronointi palvelimelle on tärkeässä osassa tukemassa offline-toiminnallisuutta, sillä sen avulla offline-tilassa tehdyt muutokset saadaan myös muiden käyttäjien saataville. Tässä web-sovelluksissa palvelin toimii synkronoinnissa aina keskiössä asiakas-palvelin-mallin mukaisesti, eivätkä eri asiakkaat synkronoi tietoja keskenään. Palvelimella sijaitsevaa dataa pidetään totuutena eikä se muutu palvelimella muuten kuin asiakkaiden lähettäessä sille päivityksiä. Koska ennen päivitysten vastaanottamista jokin toinen asiakas on voinut muuttaa dataa, jota päivitys koskee, täytyy olla jonkinlainen strategia, minkä avulla varaudutaan erilaisiin konfliktitilanteisiin synkronoinnissa.

Tietojen synkronoinnilla tarkoitetaan tässä yhteydessä käyttäjän syöttämän datan synkronointia palvelimella olevan datan kanssa. Koska samalla tilakäynnillä merkintöjä voi tehdä usea henkilö, ei yhden käyttäjän kirjauksia voida vain asettaa tilakäynnin tiedoiksi, vaan täytyy mahdollisesti yhdistää käyttäjien tekemiä kirjauksia. Lisäksi on hyvä huomioida esimerkiksi tapaus, jossa käyttäjä vaihtaa laitetta kesken käynnin. Tällöin saman käyttäjän jotkin kirjaukset saattavat jäädä laitteelle pitkäksikin aikaa ennen kuin käyttäjä seuraavan kerran kirjautuu laitteella järjestelmään. Tällöin voidaan esimerkiksi tarjota käyttäjälle mahdollisuus peruuttaa tietojen synkronointi, sillä tiedot voivat olla vanhentuneita tai tarpeettomia.

4.2 Tekniikat

Tässä luvussa käsitellään sovelluksen käyttöliittymän toteutuksessa käytettyjä tekniikoita, jotka ovat olleet läsnä offline-toiminnallisuuden liittyvän sovelluslogiikan toteuttamisessa.

4.2.1 React

ReactJS on JavaScript-kirjasto, joka on tarkoitettu käyttöliittymien toteuttamiseen. Sitä voidaankin käyttää MVC-mallia toteuttavissa sovelluksissa näkymän toteuttamiseen [30]. Sen tavoitteena on välttää näkymän turhaa päivittämistä, ja päivittää ainoastaan niitä osia, joita tarvitaan. React perustuu komponentteihin, jotka alustetaan ensin joillain parametreilla, jonka perusteella käyttäjälle renderöidään näkyviin halutut asiat. Tämän jälkeen näkymää päivitetään aina komponentin tilan muuttuessa. React huolehtii tästä päivittämisestä automaattisesti.

React tarjoaa tehokkaan ja helpon tavan käyttöliittymän päivitykseen. Niinpä sitä voi hyödyntää sovelluslogiikan toteuttamiseksi selaimessa niiltä osin, kuin verkkoyhteys ei ole välttämätöntä. React-komponenttien elinkaari on tärkeässä osassa käyttöliittymän toimintaa etenkin yhden sivun sovelluksissa, jossa päivitetään ainoastaan muuttuneet osat HTML-dokumentista. Reactin avulla muutokset voidaan päivittää automaattisesti käyttäjälle näkyviin riippuen komponentin tai sovelluksen tilasta (engl. state) ja komponentille annettavista ominaisuuksista (engl. props). Komponentin tilaa muuttamalla komponentti päivittyy ja se renderöidään uudestaan, mikäli tarpeellista. [31]

4.2.2 Redux

Redux on avoimen lähdekoodin JavaScript-kirjasto, joka toteuttaa Flux-arkkitehtuurimallia. Reduxia voi käyttää muidenkin sovelluskehysten kuin Reactin kanssa. Reduxissa on kolme pääperiaatetta sovelluksen tilan hallinointiin liittyen, jotka ovat vapaasti suomennettuna:

1. yksi totuuden lähde
2. tilaa voidaan vain lukea
3. tilamuutokset kirjoitetaan puhtaina funktioina.

Nämä tarkoittavat sitä, että sovelluksen koko tila löytyy samasta paikasta, store-komponentista. Näin kaikki sovelluksen komponentit käyttävät samaa tilaa, eikä ohjelmoijan tarvitse huolehtia muiden komponenttien tilassa tapahtuvista muutoksista, jos yhdessä komponentissa muutetaan sovelluksen tilaa. Tilaa ei voida muokata suoraan, ainoastaan action-komponentin avulla, joka sisältää tiedon siitä, mitä tapahtui. Tilaa ei voida siis muokata esimerkiksi näkymässä vaan kaikki tilamuutokset kulkevat Flux-arkkitehtuurin tiedonkulkuperiaatetta noudattaen. Tilan muutokset tapahtuvat käyttäen puhtaita eli sivuvaikutuksettomia *reducer*-funktioita. Ne ovat normaaleja JavaScript-funktioita, jotka saavat parametrinaan vanhan tilan ja palauttavat sovelluksen uuden tilan. Tämän jälkeen uuden tilan avulla voidaan päivittää kaikki näkymät.

Reduxin avulla React-komponenteille ei tarvitse ylläpitää tilaa erikseen, vaan voidaan hyödyntää sovelluksen yhteistä tilaa, jolloin tilan päivittäminen tapahtuu yhdessä paikassa. Tällöin komponenttien ei myöskään tarvitse välittää omaa tilaansa eteenpäin lapsikomponenteilleen, vaan kaikki komponentit voivat tarkkailla storen tilaa ja renderöidä sen mukaan asioita. [32]

5. RATKAISUT JA NIIDEN ARVIOINTI

Sovelluksen käyttäminen offline-tilassa vaatii kahden erityyppisen tiedon tallentamista selaimeen, joita ovat sovelluksen sisältämä data ja sovelluksen käyttämät resurssit, tässä tapauksessa JavaScript- ja tyylitiedostot sekä tietysti pohjana toimiva HTML-dokumentti. Jos selainta ei suljeta ensimmäisen latauksen jälkeen, voidaan sovellusta käyttää tallentamatta resursseja erikseen, sillä sovellus on jo ladattu ja avattu selaimessa. Sovelluksen sisältämästä datasta tarvitaan tilakäynteihin liittyvien tietojen lisäksi ainakin kirjautuneen käyttäjän tiedot.

5.1 Resurssit

Sovelluksen käyttämien resurssien tallentaminen ei ole välttämätöntä käyttäjän kannalta, jos tämä saa avattua sovelluksen valmiiksi selaimessa ennen mahdollista offline-käyttöä. Jos kuitenkin sovellus joudutaan lataamaan uudestaan, täytyy myös resurssien tallennus toteuttaa. Resurssien tallentamiseksi tarvitaan jokin muu vaihtoehto kuin esimerkiksi localStorage, jonka avulla dataa voidaan tallentaa helposti avain-arvo-pareina, mutta joka ei sovellu isompien ja monimutkaisempien tietojen tallentamiseen. Niinpä sovelluksen staattisten resurssien tallentaminen käyttäjän selaimiin päätettiin ratkaista hyödyntäen service workereitä. Seuraavaksi käsitellään tarkemmin service workerin rekisteröintiin ja asennukseen liittyviä ratkaisuja sekä resurssien käyttöä.

5.1.1 Service workerin rekisteröinti

Koska sovellus on toteutettu yhden sivun web-sovelluksena, sovellus sisältää käytännössä vain yhden HTML-dokumentin, joka toimii pohjana riippumatta käyttäjälle näytettävästä näkymästä. Tästä syystä HTML-dokumentti ladataan yhden istunnon aikana vain kerran selaimeen, mikäli käyttäjä ei päivitä sivua tai navigoi osoiterivin avulla sovelluksessa muualle. Tämä puolestaan täytyy ottaa huomioon service workerin rekisteröinnissä. Periaatteessa rekisteröinti voitaisiin tehdä vasta siinä vaiheessa, kun tilakäynti aloitetaan. Tämä tarkoittaisi käytännössä sitä, että se voidaan rekisteröidä, kun tilakäyntiin liittyvä React-komponentti alustetaan (engl. *mount*). Tällöin rekisteröintiskriptiä kutsuttaisiin aina, kun tilakäynnin kohdalle navigoitaisiin sovelluksessa, jolloin service workereita ei turhaan asennettaisi käyttäjille, jotka eivät tilakäynteihin liittyviä kirjauksia tee.

Service workeriä rekisteröidessä sille voidaan asettaa myös näkyvyysalue (engl. *scope*), jonka avulla voidaan rajata sen vaikutus vain tietyn polun alla oleviin resursseihin [33].

```
if (navigator.serviceWorker) {
    navigator.serviceWorker.register('/service-worker.js',
        { scope: '/admin/visit/' })
        .then(function (reg) {
            console.log('[Companion]',
                'Service worker registered!');
        });
}
```

Ohjelma 1. *Service workerin rekisteröinti.*

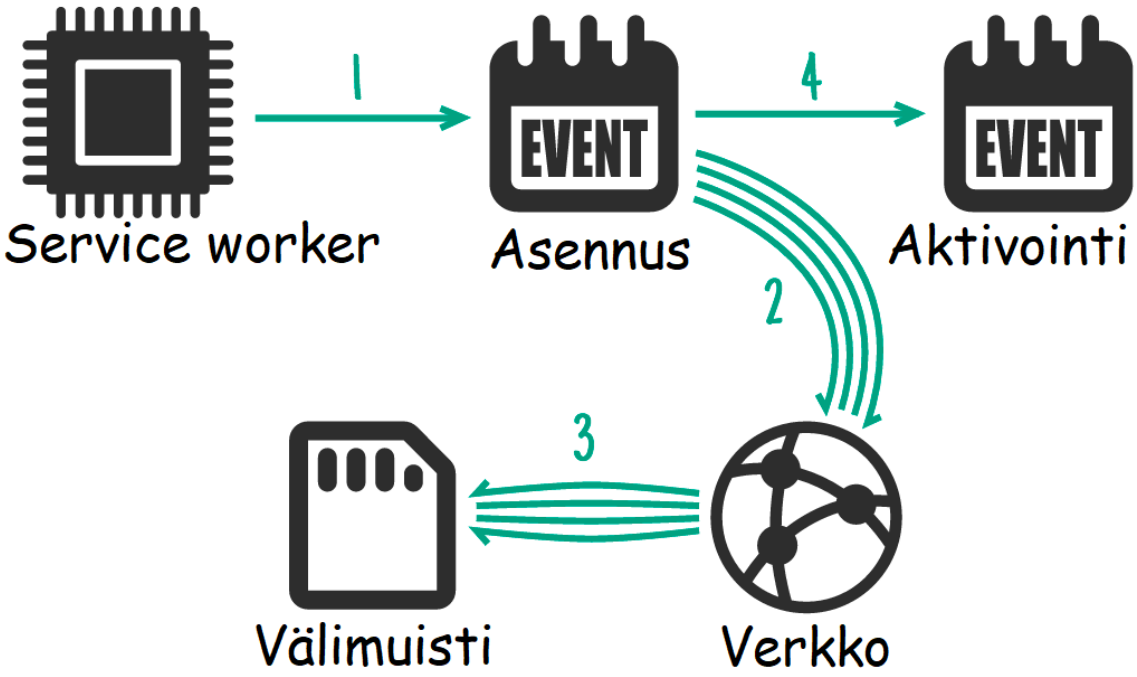
Ohjelmassa 1 on esitetty service workerin rekisteröinti näkyvyysalueen kanssa tarkastaen ensin, tukeeko selain service workereita. Tämän avulla voitaisiin määrittää, että service workerit ovat vastuussa vain tietyn polun alla olevista resursseista. Esimerkiksi voitaisiin käyttää ”/admin/visit/”-polkua, joka viittaa tässä kohtaa tilakäynteihin. Tässä tapauksessa saavutettaisiin se hyöty, että service worker ei tarttuisi esimerkiksi etusivulle kohdistuvaan pyyntöön ja tarjoaisi välimuistista offline-tilassa resursseja, joiden avulla sivu voitaisiin mahdollisesti näyttää, mutta datan puuttuessa tästä ei olisi käyttäjälle juurikaan lisäarvoa. Tällöin käyttäjälle ei turhaan tarjottaisi mielikuvaa siitä, että sovellus toimii offline-tilassa semmoisten osien kohdalla, joista dataa ei ole tallennettu lokaalisti.

5.1.2 Service workerin asennus

Service workerin asennusprosessin kulku on esitetty kuvassa 11, josta huomataan, että service worker aktivoidaan vasta resurssien latauduttua, mikäli sen toimivuus on riippuvainen ladattavista resursseista. Muussa tapauksessa voitaisiin aloittaa aktivointi myös kesken latauksen. Jos tässä kohtaa jonkin resurssin tallennus epäonnistuu, asennus on epäonnistunut, jolloin service workeriä ei aktivoida. Tässä ei toisaalta ole mitään resursseja, mitä tarvittaisiin ainoastaan myöhemmin, joten siinä mielessä ei voida tehdä niin, että joitakin resursseja ladattaisiin vasta myöhemmin. Asennusta seuraava vaihe on siis aktivointi, jonka aikana vanhat resurssit poistetaan, mikäli service workerin versio ollaan päivitetty uuteen. Versio annetaan service workerin määrittelevässä service_worker.js.erb-tiedostossa. Se ei päivity dynaamisesti, joten jos service worker halutaan päivittää käyttämään uutta versiota, täytyy tiedostoon päivittää uusi versionumero. Asennusskripti on lisäksi esitetty liitteessä A, josta käy ilmi kaikki asennusvaiheessa ladattavat resurssit ja lisäksi staattisten resurssien versionhallinta.

Aiemmin mainittiin jo, että sovelluksen luonteesta johtuen sovellus ei sisällä useita HTML-dokumentteja, jotka pitäisi service workerin asennusvaiheessa ladata selaimen välimuistiin. Asennettaessa ladataan siis välimuistiin sama HTML-dokumentti, joka toimii

sovelluksen pohjana. Lisäksi käyttöliittymään liittyvät JavaScriptit ladataan tässä vaiheessa. JavaScript-resursseja on kahdessa paikassa. Ensinnäkin joitakin yleisesti käytettäviä skriptejä tarjotaan Railsissä oletuksena olevan asset-liukuhihnan avulla. Tämän lisäksi hyödynnetään Railsin Webpacker-kirjastoa [34] käyttöliittymäkoodin pakkaamiseen. Webpacker tuottaa yhden tiedoston, joka sisältää sovelluksen perusrakenteen. Lisäksi loput tiedostot pakataan palasina omiksi kokonaisuuksikseen, jotka ladataan tarpeen mukaan. Tämän ansiosta sovelluksen koon kasvaessa ei tarvitse ladata liian suurta osaa käyttöliittymäkoodista kerralla, vaan vain se osa, mitä tarvitaan. Tämä osiin jakaminen tuottaa kuitenkin haasteen asennusvaiheessa, sillä tilakäynteihin liittyvän palasen lataamiseksi pitäisi tietää, missä palassa se on. Vaihtoehtoisesti voitaisiin ladata kaikki palaset.



Kuva 11. *Servicen workerin asennus [21].*

Railsin asset-liukuhihnan avulla saadaan myös sovelluksen tyylit pakattua yhdeksi CSS-tiedostoksi [35], joka voidaan ladata asennusvaiheessa. Tyyleistä on kuitenkin huomiotava, että ulkoisten kirjastojen tyylejä käytettäessä ne eivät ole samassa asset-kansiossa, jolloin ne on ladattava erikseen, mikäli ei haluta kopioida isoja tiedostoja asset-kansioon. Tällöin on huomattava, että näiden kirjastojen päivittyessä pitäisi nämä tiedostot myös päivittää erikseen. Tämä ongelma ratkaistiin siten, että näiden tyylien import-komento voidaan laittaa osaksi React-tiedostoa, jossa niitä käytetään. Tällöin ei tarvitse erikseen ladata niitä asennusvaiheessa. Tällöin ei tule riskiä siitä, että jonkin tiedoston lataaminen unohtuisi tässä kohtaa. Webpacker hoitaa näin ollen tyylien lataamisen, kun kyseisiä tyylejä tarvitaan.

Näiden lisäksi tässä vaiheessa ladataan kuvia ja fontteja, joita on käytetty tyylitiedostoissa. Tarvittavia kuvia on sovelluksen tausta sekä yläpalkissa näkyvä asiakkaan logo. Fontteja puolestaan käytetään erilaisten ikonien näyttämiseen käyttäjälle. Näiden lisäksi sovelluksen sisältämät käännökset eri kielille tarvitaan, jotta näkymässä voidaan näyttää kaikki tekstit valitulla kielellä. Käännökset generoidaan palvelimella JSON-muodossa ja tämä vastaus voidaan tallentaa välimuistiin, jotta käännöksiä voidaan käyttää, vaikka palvelimeen ei saataisi yhteyttä. Käännökset eivät välttämättä ole enää staattinen resurssi, mikäli niiden muokkaaminen mahdollistuu esimerkiksi jonkin toisen palvelun kautta. Tällöin pitäisi käännökset pystyä päivittämään aina välillä ilman, että staattisten resurssien versiota päivitetään.

5.1.3 Resurssien käyttö

Välimuistiin tallennetut resurssit voidaan hakea suoraan sieltä, jolloin ei tarvitse odottaa verkon vastausta, mikäli yhteys on hidas. Mikäli kyseessä on pyyntö, jota ei ole tallennettu välimuistiin, suoritetaan se verkkoa käyttäen. Tämä prosessi on esitetty ohjelmassa 2, jossa rivillä 4 haetaan ensin vastausta välimuistista. Tämän jälkeen palautetaan vastaus rivillä 7, mikäli se löytyi. Tämän jälkeen mahdollisiin navigointipyyntöihin vastataan palauttamalla sovelluksen pohjana toimiva HTML-dokumentti, mikä tehdään rivillä 19. Lopuksi haetaan vielä vastaus verkkoa käyttäen rivillä 23, jos vastausta ei voitu palauttaa välimuistista. Tämä prosessi vastaa myös aliluvun 3.3.3 kuvassa 7 esitettyä resurssien hakua välimuistista käyttäen verkkoa varalla.

Vaihtoehtoinen tapa resurssien tarjoamiseen käyttäjälle olisi hakea ensin vastausta verkosta ja sen jälkeen välimuistista, jos verkosta ei vastausta saada. Tällöin online-tilassa varmistuttaisiin resurssien ajantasaisuudesta, mutta hitaan yhteyden tapauksessa jouduttaisiin odottamaan vastausta mahdollisesti kauankin. Suorituskykysyistä välimuistin käyttö ensisijaisena lähteenä staattisten resurssien kohdalla onkin paras ratkaisu. Tällöin täytyy service workerin tallentamat resurssit päivittää, kun ne muuttuvat, service workerien asennusvaihetta käsittelevässä aliluvussa mainitulla tavalla.

```
// Suoritetaan, kun selain tekee pyynnön
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      // Jos välimuistista löytyy vastaus, palautetaan se.
      if (response) {
        return response;
      }

      // Jos välimuistista ei löydy vastausta, palautetaan navigoin-
tipyynnöille vastauksena sovelluksen pohjana toimiva HTML-dokumentti.
      if (event.request.mode === 'navigate' ||
      (event.request.method === 'GET' &&
      event.request.headers.get('accept').includes('text/html'))) {
           console.log('[Serviceworker]',
               "Fetching offline content",
               event);

           // offline.html
           return caches.match('/offline');
      }

      // Jos kyseessä ei ole navigointi, yritetään käyttää verkkoa.
      return fetch(event.request);
    })
  );
});
```

Ohjelma 2. *Resurssien haku välimuistista verkon ollessa varalla.*

5.1.4 Arviointi ja vaihtoehdot

Staattisten resurssien tallentamiseen service worker on hyvä ratkaisu, sillä sovelluksen tarvitsemat resurssit on helppo tallentaa heti asennusvaiheessa. Tämän jälkeen ne voidaan tarjota käyttäjälle myös välimuistista, vaikka verkkoyhteys olisikin saatavilla, sillä ne muuttuvat harvoin, ja hitaan verkkoyhteyden tapauksessa palvelimen vastausta voi joutua odottamaan. Service workereistä on kuitenkin huomioitava, että niiden avulla voi alkaa tallentamaan välimuistiin resursseja vasta, kun ne on asennettu ja aktivoitu käytettäviksi. Niinpä sivun ensimmäisen latauksen yhteydessä asennettava worker ei käsittele saman pyynnön mukana ladattavia resursseja eikä pysty niitä tallentamaan.

Lisäksi on hyvä huomata, että tällä hetkellä Internet Explorer ei tue service workereita. Tällöin sovelluksen uudelleen lataaminen offline-tilassa ei onnistu IE:llä, jolloin sen käyttäjät joutuvat tyytymään huomattavasti rajoitetumpaan offline-tukeen. Asiakkaan puolelta vaatimusta selaintuesta kaikille selaimille ei ole. Lisäksi heillä on tavoitteena päästä IE:n käytöstä eroon tulevaisuudessa. Tästä syystä erillistä ratkaisua vain IE:tä varten ei olla sovelluksen oflline-toiminnallisuuden takaamiseksi toteuttamassa. Service worker on vielä kohtuullisen moderni ja kehittyvä tekniikka, joka on kasvattanut suosiotaan viime

aikoina. Tällä hetkellä globaalista selainten markkinaosuudesta hieman yli 90% tukee service workereita whatwebcando.today-verkkosivuston mukaan [36]

Koska kyseessä on yhden sivun sovellus, käytännössä pyyntöjä tehdään vain, kun sivua päivitetään, tai navigoidaan osoiterivin avulla. Sovelluksen käyttöliittymässä tehtävät navigoinnit käsitellään Reactin avulla, jolloin näkymän päivittäminen tapahtuu selaimessa eikä palvelimelle lähde pyyntöä. Tämä on ristiriitainen asia käyttökokemuksen kannalta, koska sovelluksessa pystytään offline-tilassa navigoimaan eri sivuilla, mutta sivun päivittäminen tuottaa virheilmoituksen verkkoyhteyden puuttumisesta muiden kuin tilakäyntiin liittyvien näkymien kohdalla. Tästä syystä käytettävyyden kannalta voisi olla parempi, ettei näkyvyysaluetta rajata kovin tarkasti, koska tarvittavat resurssit sovelluksen näyttämiseen on joka tapauksessa ladattu, sillä nämä sisältyvät koottuun JavaScript-tiedostoon. Tämä voitaisiinkin ratkaista siten, että käyttäjälle näytettäisiin offline-tilassa ilmoitus, jossa kerrottaisiin, että ainoastaan tilakäynteihin liittyvä data on saatavilla eikä muihin näkymiin navigoidessa voida näyttää dataa.

Varteenotettavia vaihtoehtoja service workereiden toiminnallisuuden toteuttamiseksi ei juuri ole. Aiemmin käytössä ollut application cache on vanhentunut ja sen tuki on poistumassa [37]. Sen avulla saataisiin tosin offline-toiminnallisuus IE:lle, mutta toisaalta ajan käyttäminen IE-tuen tarjoamiseksi ei tämän projektin puitteissa ole välttämättä kovin hyödyllistä. Offline-plugin [38] tarjoaisi mahdollisuuden myös Application Cachen käyttöön service workereiden ohella. Lisäksi sen avulla voitaisiin mahdollisesti yksinkertaistaa ja parantaa service workereiden tarjoamaa toteutusta webpackia käyttävissä sovelluksissa. Sen käyttö voisi tulla myös kyseeseen, mikäli service workereitä tarvittaisiin useampaan eri toiminnallisuuteen sovelluksessa. Tässä sovelluksessa on toki hyödynnetty myös asset-liukuhihnaa webpackerin ohella, joten nämä resurssit pitäisi määritellä erikseen ladattaviksi offline-pluginia käytettäessä.

5.2 Tietojen tallennus

Pelkän kyseiseen tilakäyntiin liittyvän datan lisäksi sovelluksen tarvitsemaa dataa ovat esimerkiksi kirjautuneen käyttäjän tiedot ja oikeudet. Näitä tietoja ei tallennetta mihinkään staattiseen tiedostoon, vaan ne ladataan selaimeen, kun käyttäjä käyttää sovellusta. Käyttäjän tiedot tarvitaan luonnollisesti sovelluksessa heti kirjautumisen jälkeen ja tilakäyntiin liittyvät tiedot, kun tilakäynteihin navigoidaan sovelluksessa. Tietojen tallennuskohteena käytetään localStoragea siitä syystä, että sen antama tallennustila riittää tilakäynteihin liittyvän datan tallentamiseen.

5.2.1 Sovelluksen tilan hallinta

Sovelluksen datan tallentamisen lisäksi täytyy huomioida se sovelluksen tilan hallinta. Jotta ei tarvitsisi ylläpitää erikseen tallennettuja tietoja localStoragessa sekä sovelluksen tilaa selaimessa, tähän otettiin avuksi Redux-kirjasto, jonka avulla sovelluksen tila saadaan keskittyä yhteen storeen. Tällöin tilakäynteihiin liittyvien tietojen käsittelyyn käytettävien React-komponenttien ei tarvitse jokaisen hallinoida itse omaa tilaansa. Keskitetyn tilan tallentaminen offline-tilassa käytettäväksi onnistuu hyödyntäen redux-offline -kirjastoa, joka on itsenäinen offline-toiminnallisuuden toteuttamista helpottava työkalu [39]. Se puolestaan hyödyntää redux-persist-kirjastoa sovelluksen tilan tallentamiseen selaimeen [40]. Oletuksena kohteena käytetään localStoragea, mutta tarvittaessa voidaan konfiguroida tallennuskohteeksi myös muita vaihtoehtoja. Tällöin sovelluskehittäjän ei tarvitse huolehtia tietojen tallentamisesta selaimeen tai lukemisesta selaimesta, sillä tilan muuttuessa se päivitetään automaattisesti myös tallennuskohteeseen.

Lisäksi redux-offline päivittää sovelluksen tilaa sen mukaan, onko verkkoyhteys saatavilla. Tämä helpottaa, kun sovelluslogiikassa ei tarvitse erikseen tarkastella verkkoyhteyden saatavuutta. Toinen merkittävä hyöty on se, että sovelluksen aktioille voidaan antaa pyynnön onnistumisen tai epäonnistumisen seurauksena suoritettava aktio. Tämän ansiosta voidaan esimerkiksi palvelinpyynnön epäonnistuessa palata sovelluksessa edelliseen tilaan, jos esimerkiksi tallennus ei onnistunut.

```
// Aktio, jolla haetaan terveyriskit.
const getHealthRisks = () => ({
   type: Types.LIST_HEALTH_RISKS_REQUEST,
   meta: {
       // Redux-offlinen hyödyntämät tiedot
       offline: {
           // Aktion suoritukseen liittyvä pyyntö
           effect: { url: Routes.health_risk_list_path() },
           // Onnistuneen suorituksen jälkeen suoritettava aktio.
           commit: { type: Types.LIST_HEALTH_RISKS_SUCCESS }
       }
   }
});
```

Ohjelma 3. *Terveysriskien hakuun käytettävän aktion luominen.*

Datan tallentaminen selaimeen tapahtuu siis redux-kirjastoja hyödyntäen silloin, kun se ensimmäisen kerran ladataan palvelimelta selaimeen ja asetetaan sovelluksen tilaan reducer-funktion avulla. Redux-offlinen hyödyllisyys tulee ilmi myöhemmillä latauskerroilla, kun komponentin alustuksessa tehdään pyyntö palvelimelle tarvittavan datan näyttämiseksi käyttäjälle. Pyynnön onnistuessa sovelluksen tilaa päivitetään asianmukaisesti, jolloin haetut tiedot renderöityvät käyttäjälle. Jos pyyntöä ei voida suorittaa tai se

epäonnistuu, käytetään automaattisesti Reduxin storeen paikallisesti tallennettua tilaa. Tästä on myös se etu, että käyttäjälle voidaan näyttää heti tietoja, vaikka lataus olisi kesken. Tällöin hidas verkkoyhteys ei estä käyttäjää käyttämästä sovellusta. Tämän toiminnallisuuden toteuttava aktio on määritelty ohjelmassa 3, josta nähdään, kuinka redux-offlinelle annetaan onnistuneen pyynnön jälkeen suoritettava aktio rivillä 10 *commit*-attribuuttia hyödyntäen. Vastaavasti voitaisiin antaa epäonnistuneen pyynnön jälkeen suoritettava aktio *rollback*-attribuuttia käyttäen. Tallennetun tilan hakeminen paikallisesti tapahtuu Reduxissa automaattisesti, joten tälle pyynnölle ei tarvitse epäonnistuneen haun jälkeistä toimintoa määritellä.

Kun tilakäynnin aikana kirjataan havainto sovelluksessa, tallennetaan syötetyt tiedot luonnollisesti sovellukseen tilaan. Lisäksi yritetään tehdä pyyntö palvelimelle tietojen päivittämisestä. Jos redux-offline havaitsee verkkoyhteyden puuttumisen, pyyntö jää odottamaan yhteyden palautumista. Tällöin pyynnön lähetys säilyy redux-offlinen vastuulla eikä siitä tarvitse huolehtia. Kun pyyntöön saadaan vastaus, voidaan sen lopputuloksesta riippuen reagoida pyyntöön jollain aktiolla. Onnistuneen pyynnön tapauksessa sovelluksen tilaa ei todennäköisesti tarvitse muuttaa, sillä se on päivitetty jo ennen pyynnön lähettämistä, jotta sovelluksen käyttöä voidaan jatkaa offline-tilassa riippumatta siitä kauanko vastauksen saamisessa kestää. Mikäli pyyntö epäonnistuu, voidaan sovelluksen tila palauttaa ennalleen ja ilmoittaa käyttäjälle, ettei muutosten tallentaminen onnistunut.

5.2.2 Arviointi ja vaihtoehdot

LocalStoragen käyttökelpoisuus tallennuskohteena riippuu hyvin pitkälti tallennettavan datan määrästä, sillä siihen mahtuu verrattain varsin vähän dataa (n. 5 Mt selaimesta riippuen [41]), ja monimutkaisten tietorakenteiden tallentaminen voi olla ongelmallista. Myöskään tiedostojen tallennus ei tällöin onnistu. Tietojen tallennuskohteen vaihtamista voitaisiin harkita, jos myöhemmin halutaan tallentaa enemmän dataa käytettäväksi offline-tilassa. Tällöin esimerkiksi IndexedDB:n käyttäminen ensisijaisena kohteena olisi hyvä vaihtoehto, sillä suuremman tallennustilan lisäksi se tukee tiedostojen tallentamista. Tässä tapauksessa todennäköisimmin valittaisiin localForage sovelluksen tilan store-komponentin tallennuskohteeksi, jolloin tarjottaisiin myös rajoitettu käyttömahdollisuus IndexedDB:tä tukemattomille selaimille IndexedDB:n ollessa oletusvaihtoehto.

LocalStoragen kohdalla on myös hyvä harkita tallentamisen turvallisuutta. Tässä voitaisiin harkita esimerkisi tietojen salaamista.

Sovelluksen dataan liittyviä pyyntöjä voitaisiin myös tallentaa välimuistiin service workereitä hyödyntäen. Tällöin ne olisivat haettavissa välimuistista eikä niitä tarvitsis tallentaa

esimerkiksi localStorageen. Nämä pyynnöt voitaisiin tällöin tehdä jo service workerin asennusvaiheessa, jolloin ei tarvitsisi odottaa, että pyyntöjen tallentumista käyttäjän navigoidessa sovelluksessa. Tässä on kuitenkin heikkoutena se, että POST-pyyntöjä ei voida samalla tavalla tallentaa välimuistiin kuin GET-pyyntöjä, sillä ne muuttavat palvelimen tilaa. Tällöin offline-tilassa tapahtuneet datan muokkaukset pitäisi huomioida erikseen eikä voitaisi luottaa välimuistista tulevaan tiedon ajantasaisuuteen. Siitä syystä tämä sopisi ratkaisuksi vain, jos dataa ei muokattaisi.

Käytettäviä kirjastoja valitessa on hyvä huomioida, että esimerkiksi Redux voi olla yksinkertaisten sovellusten kohdalla liian monimutkainen, jotta sen käyttö olisi järkevää. Tällöin voidaan joutua tekemään kompromisseja sen suhteen, halutaanko offline-toiminnallisuudessa hyödyntää valmista kirjastoa ja kasvattaa sovelluksen kompleksisuutta, vaiko pitää sovelluslogiikka yksinkertaisena, mutta toteuttaa itse tietojen sovelluksen tilan tallentaminen lokaalisti. Toisaalta esimerkiksi verkkopyyntöihin liittyvä valmis toiminnallisuus voi helpottaa huomattavasti palvelinpyyntöihin liittyvää logiikkaa.

5.3 Tietojen synkronointi

Tietojen synkronoinnin toteuttaminen on huomattavasti moniulotteisempaa kuin tietojen tallentaminen. Siinä mielessä synkronoinnin toteutuksen kuvaamisessakaan ei ole niin mielekästä keskittyä pieniin yksityiskohtiin erilaisten kenttien arvojen vertailussa, vaan kuvailla joitain pääperiaatteita, joita synkronointiin liittyvässä tietojen vertailussa ja konfliktien ratkaisussa on käytetty. Ensinnäkin sovelluksen tietokannassa tilakäyntiin liittyvät taulut sisältävät aikaleiman viimeisimmästä päivityksestä sekä päivitykseen tehneen käyttäjän tunnisteen. Kun tietoja haetaan palvelimelta selaimessa tallennusta varten, voidaan näitä tietoja käyttää vertailussa selaimessa tehtyihin muutoksiin.

5.3.1 Pyynnöt palvelimelle

Kun redux-offlinea hyödynnetään aktioita luotaessa, kaikki verkkopyynnöt lisätään automaattisesti sen ylläpitämään jonoon, josta pyyntöjä suoritetaan vain online-tilassa. Tällöin pyynnöt suoritetaan saman tien, jolloin jonon toiminnallisuudella ei ole suurta merkitystä, koska pyyntöjä ei ehdi kertyä odottamaan. Offline-tilassa jonoa voidaan kuitenkin hyödyntää muokkaamalla ja sulauttamalla jonossa olevia pyyntöjä, mikäli uudet pyynnöt koskevat samaa alkiota. Tämä voidaan toteuttaa korvaamalla redux-offlinen pyyntöjä jonoon lisäävä *enqueue*-metodi muokatulla versiolla smart-queue-pluginin [42] vastaavasta. Muokatun metodin ohjelmakoodi on esitetty liitteessä B. Se toimii siten, että aktion *meta*-attribuutissa annetaan *queue*-objekti, joka voi sisältää pyyntöön liittyvän metodin, tunnisteen, näkyvyysalueen ja tiedon siitä, onko kyseessä väliaikainen tunniste. Tästä

on esimerkki ohjelman 4 riveillä 23-24. Mikäli pyynnölle ei ole määritelty *queue*-attribuuttia, suoritetaan oletustoiminnalisuus eli lisätään pyyntö jonoon riippumatta pyynnön sisällöstä, mikä on suoritettu koodissa rivillä 15.

```
const createHealthRisk = (token, description, strId, ...) => ({
   // Aktio, joka suoritetaan heti, kun terveysriskiä yritetään
luoda sovelluksessa.
   type: Types.CREATE_HEALTH_RISK_REQUEST,
   meta: {
       description: description,
       // Väliaikainen tunniste
       strId: strId,
       // Muut kentät
       ...,
       offline: {
           // Palvelinpyyntö
           effect: {
               url: Routes.health_risk_create_path(),
               method: 'POST',
               body: JSON.stringify({
                   authenticity_token: token,
                   health_risk: { description: description, ... }
               })
           },
           // Onnistuneen pyynnön jälkeen suoritettava aktio
           commit: {
               type: Types.CREATE_HEALTH_RISK_SUCCESS,
               meta: { strId: strId }
           },
           // Epäonnistuneen pyynnön jälkeen suoritettava aktio
           rollback: {
               type: Types.CREATE_HEALTH_RISK_FAILURE,
               meta: { strId: strId }
           },
           // Pyynnön jonoon lisäämiseen liittyvät tiedot
           queue: {
               method: QueueActionTypes.CREATE,
               key: strId,
               scope: 'health_risk_write'
           }
       }
   })
});
```

Ohjelma 4. *Terveysriskin luomiseen käytettävän aktion luominen.*

Aiemmin mainittiin, että käyttäjän tehdessä tilakäyntiin liittyvän kirjauksen tiedot tallennetaan heti sovelluksen tilaan käyttäjälle näyttämistä varten. Tähän liittyvän aktion luominen on esitetty ohjelmassa 4, jossa rivillä 4 määritetään tiedot paikallisesti tallentava aktio. Rivillä 13 on palvelinpyyntö kirjauksen päivittämiseksi, joka siirtyy jonoon odottamaan verkkoyhteyttä, jotta tiedot voitaisiin synkronoida palvelimelle. Jos kyseessä on

esimerkiksi uuden terveydenvaaran luominen, voidaan pyyntö lähettää ja käsitellä sellaisenaan. Kun pyyntöön saadaan vastaus, suoritetaan lopputuloksesta riippuen joko *commit*- tai *rollback*-toiminto. Onnistuneen luonnin jälkeen päivitetään tietokannasta saatu tunniste (ID) paikalliselle alkiolle, jotta siihen liittyvät operaatiot voidaan yhdistää palvelimen alkioon. Tällöin korvataan paikallisesti generoitu väliaikainen tunniste, jota käytetään operaatioiden tunnisteena ennen sitä. Epäonnistuneen pyynnön seurauksena poistetaan luotu terveydenvaara paikallisista tiedoista ja näytetään käyttäjälle virheilmoitus.

Offline-tilassa luotua terveydenvaaraa muokatessa tälle ei luonnollisesti ole olemassa tietokantatunnistetta, mikäli tietoja ei ole vielä synkronoitu palvelimelle. Tällöin päivittäminen voidaan toteuttaa paikallisiin tietoihin normaalisti, mutta lähteviin pyyntöihin ei voida lisätä olemattoman tunnisteen omaavaa päivityspyyntöä. Tämä ongelma on ratkaistu *enqueue*-metodin toiminnallisuudessa. Jos ollaan lisäämässä jonoon päivitystä sellaisella tunnisteella, jolla on jo jonossa luomispyyntö, voidaan luomispyynnön sisältö korvata päivityspyynnön sisällöllä ja jättää päivitys tekemättä. Tällöin pyynnön mennessä palvelimelle tietue luodaan samoin tiedoin, kuin mitä paikalliseen tietueeseen on päivitetty.

Päivitettäessä jo palvelimella olemassa olevia tietoja voidaan jonoa hyödyntää siten, että mahdolliset uudet samaa terveydenvaaraa koskevat pyynnöt sulautetaan jonossa jo olevan pyynnön kanssa samaan. Tällöin ei turhaan ensin päivitetä vanhentuneita tietoja palvelimelle ja sen jälkeen uusia päivitettyjä tietoja. Lisäksi tällä pyritään varmistamaan, ettei samaan terveydenvaaraan liittyviä pyyntöjä ole jonossa useita, pois lukien lukuoperaatiot, jotka eivät muuta palvelimen tilaa. Kun jonossa on vain yksi tietuetta muuttava operaatio, on seuraavan pyynnön sulauttaminen huomattavasti yksinkertaisempaa.

Tietoja poistettaessa voidaan jonosta poistaa samaa terveydenvaaraa koskeva mahdollinen luonti -tai muokkauspyyntö. Jos luontipyyntö ei ole mennyt palvelimelle asti, on sitä turha sinne lähettää, mikäli alkio poistetaan selaimessa. Tällöin myöskään poistopyyntöä ei jonoon lisätä, koska palvelimella ei kyseistä alkiota ole. Poistettaessa tietokannasta löytyvää alkiota voidaan jonosta poistaa siihen liittyvät päivityspyynnöt, sillä niillä ei ole poiston jälkeen merkitystä ja näin voidaan välttyä mahdollisilta konflikteilta, mikäli päivitys aiheuttaisi konfliktitilanteen, kun palvelimen data olisi muuttunut välillä. Tässä on kuitenkin huomioitava, että poiston epäonnistuessa paikallinen tila on palauttettu ennalleen. Tämän jälkeen palvelimen ja selaimen tiedot eivät vastaa toisiaan, sillä päivityspyynnön yhteydessä tehdyt muutokset eivät ole siirtyneet palvelimelle.

Näiden tilanteiden lisäksi voisi esimerkiksi jossain virhetilanteessa poisto-operaation jälkeen jonoon tulla toinen poisto- tai päivitysoperaatio. Tällöin ei kuitenkaan aiheudu haittaa myöhemmistä operaatioista, sillä alkio pysyy poistettuna tietokannassa, vaikka sitä muokattaisiin jälkikäteen.

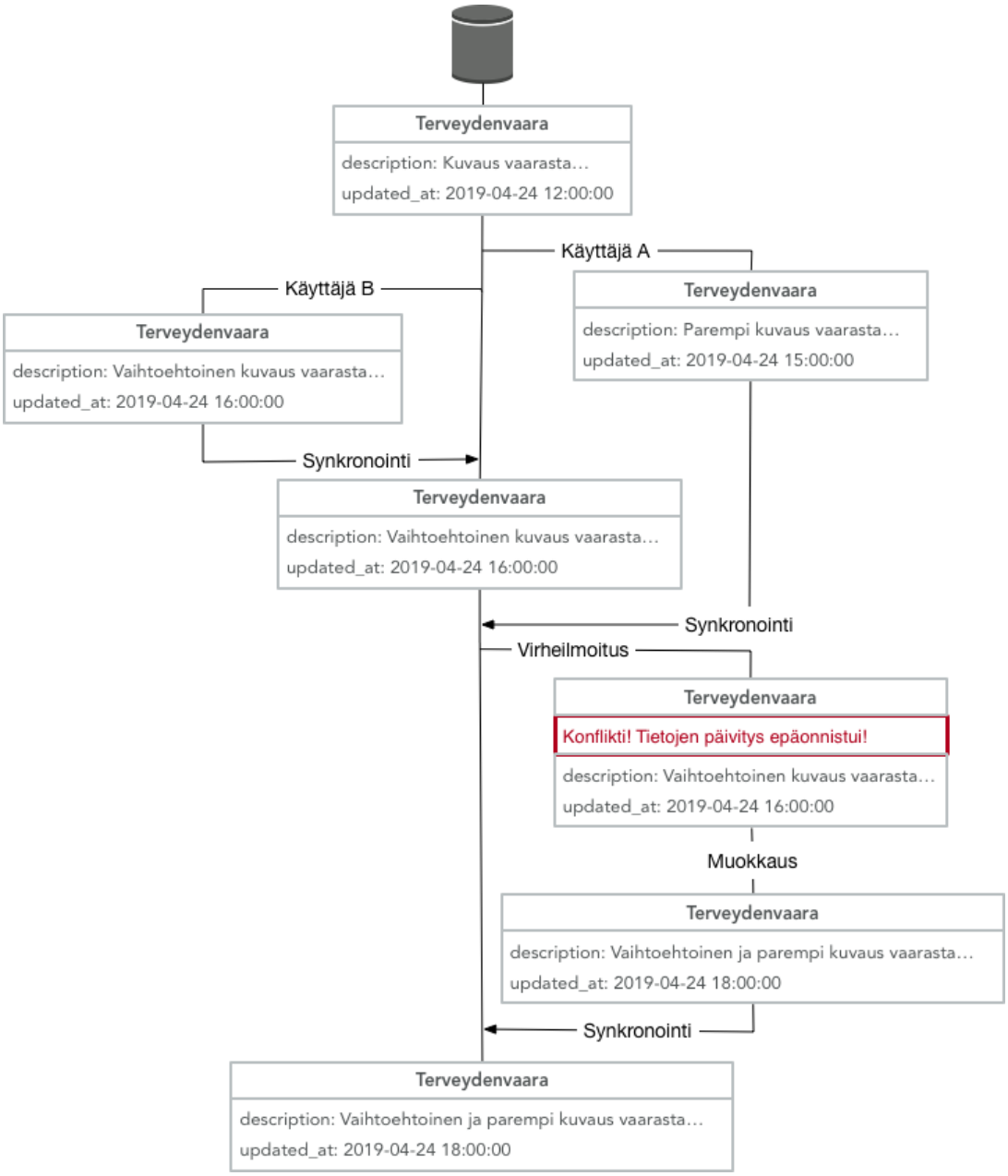
Kun verkkoyhteys saadaan käyttöön offline-tilassa työskentelyn jälkeen, on jonoon mahdollisesti kertynyt useampia pyyntöjä, joita aletaan suorittamaan, mikäli sovellus on käynnissä. Jos käyttäjä kuitenkin on sulkenut selaimen tekemiensä muutosten jälkeen, aloitetaan tietojen synkronointi vasta, kun tämä avaa sovelluksen uudelleen. Jotta paikalliset muutokset eivät jää synkronoimatta, näytetään käyttäjälle käyttöliittymässä tieto synkronoimattomista muutoksista. Lisäksi käyttäjälle voidaan huomauttaa, että muutosten päivittyminen vaatii uudelleenkirjautumisen, mikäli tämä on kirjautumassa ulos, navigoimassa pois sovelluksesta tai sulkemassa selainta. Tällöin käyttäjän vastuulla on varmistaa, että muutokset tulee synkronoitua myös palvelimelle, kun käyttäjällä on jälleen verkkoyhteys saatavilla. Tulevaisuudessa voitaisiin myös harkita synkronoimattomista muutoksista ilmoittamista käyttäjälle näytettävällä push-notifikaatiolla, jonka service workerit mahdollistavat. Tällöin voitaisiin muistuttaa käyttäjää kirjautumaan sovellukseen, vaikka selain olisi suljettuna.

5.3.2 Konfliktien havaitseminen ja ratkaisu

Kun käyttäjä A tekee selaimessa muutoksen olemassa olevaan terveydenvaaraan, esimerkiksi muuttaa sen sanallista kuvausta, päivitetään muuttuneiden tietojen lisäksi alkion *updated_at*-aikaleima, joka kertoo tiedon päivitysajankohdan. Kun pyyntö lähtee jonosta palvelimelle, lähetetään aikaleima samalla mukana. Palvelimella tapahtuvassa tietojen validoinnissa verrataan tällöin aikaleimaa tietokannan vastaavan kentän arvoon. Mikäli tietokannan arvo on aiempi kuin selaimen lähettämä tieto, on lähetetty muutos oletettavasti viimeisin tieto. Tällöin voidaan tiedot päivittää, mikäli muutkin validoinnit onnistuvat. Jos taas palvelimen aikaleima on selaimen vastaavaa myöhäisempi, on käyttäjä B päivittänyt tietoja sen jälkeen, kun käyttäjä A teki omat muutoksensa. Tällöin tietoja ei päivitetä ja A:lle näytetään virheilmoitus päivityksen epäonnistumisesta konfliktista johtuen. Tämä siitä syystä, että A:n muutokset ovat mahdollisesti olleet selaimen muistissa kauankin ja tiedot muuttuneet tämän jälkeen.

Tässä tapauksessa konfliktin ratkaisussa uusin tieto voittaa tallennusajankohdasta mitattuna eikä pyynnön lähetysajankohdalla ole merkitystä. Tämä ratkaisu valittiin, koska saman alkion muokkaaminen on harvinainen tapaus ensinnäkin siitä syystä, että tilakäyneillä havaintoja kirjaavat todennäköisesti muokkaavat omia havaintojaan. Lisäksi of-

fline-tilassa työskennellessään samaan tilakäyntiin liittyvät henkilöt voivat todennäköisesti olla vuorovaikutuksessa keskenään, jolloin saman alkion samanaikainen muokkaaminen ei ole kovin todennäköistä. Tämänkaltainen tilanne, jossa kaksi henkilöä muokkaa samaa lomaketta samanaikaisesti voi tuottaa ongelmia kaikissa web-sovelluksissa, myös online-tilassa, jos tätä mahdollisuutta ei ole huomioitu. Tämä johtuu siitä, että yleensä lomakkeen sisältämät tiedot tallennetaan sellaisenaan, jolloin ensimmäisenä tallentaneen tiedot kirjoitetaan yli.



Kuva 12. *Konfliktin ratkaiseminen, kun aiemmin tallennetut muutokset synkronoidaan myöhemmin.*

Konfliktien ratkaisuun liittyvää toiminnallisuutta voitaisiin tulevaisuudessa parantaa lisäämällä kenttäkohtainen aikaleima, jolloin voitaisiin päivittää muutokset, mikäli kyseistä

kenttää ei olisi päivitetty muutosten tallentamisen jälkeen. Tässä tosin voisi tulla ongelmia, mikäli useamman kentän arvot ovat jollain tavalla riippuvaisia toisistaan. Toinen mahdollinen parannus olisi näyttää käyttäjälle konfliktin aiheuttaneet tiedot ja tarjota mahdollisuus niiden lähettämiseen ratkaisemiseen. Tällöin voisi myös muokata paikallisia muutoksia ennen lähettämistä ja sisällyttää molempien käyttäjien muutokset esimerkiksi tekstikentän sisältöön. Tällöin käyttäjä olisi vastuussa siitä, etteivät uusimmat päivitykset katoa, mutta saisi myös omat havaintonsa mukaan.

Kuvassa 12 on esitetty tämänkaltaisen konfliktin havaitsemiseen ja ratkaisuun liittyvän prosessin kulku. Molemmat käyttäjät siis muokkaavat samassa tilassa olevaa alkiota, mutta vaikka käyttäjä A tekee havaintonsa aiemmin, synkronoidaan hänen tekemänsä muutokset käyttäjä B:n muutosten jälkeen. Tässä tapauksessa A muokkaa kuvauksen sisällön sisällyttämään myös B:n tekemät havainnot. Tällöin tehdään uusi päivitys, jonka seurauksena myös aikaleima päivittyy myöhäisemmäksi. Vaihtoehtoisesti käyttäjä A voi jättää tallentamatta uudestaan, jolloin B:n tekemät muutokset jäävät voimaan. Tämän voi siis nykyiselläkin toiminnallisuudella tehdä, mutta luonnollisesti käyttäjän täytyy kirjoittaa tekemänsä havainnot uudelleen, sillä niitä ei pystytty tallentamaan.

5.3.3 Arviointi ja vaihtoehdot

Synkronointivaiheessa voi tulla ongelmia, kun alkion luontioperaatio on kesken ja yritetään suorittaa kirjoitusoperaatioita alkion väliaikaista tunnistetta käyttäen. Tämän pitäisi toki olla hyvin harvinainen tilanne, joka vaatisi todella hidasta verkkoyhteyttä, mutta kuitenkin sen olemassa oloa, jotta pyyntöjen suoritus jonosta voidaan aloittaa. Tällöin jälkimmäistä pyyntöä ei voida palvelimelle lähettää, joten käyttäjälle voitaisiin näyttää virheilmoitus tai vaihtoehtoisesti odottaa luomispyynnön valmistumista, jonka onnistuessa voitaisiin päivityspyyntö lähettää käyttäen oikeaa tunnistetta.

Yksi mahdollinen lisätoiminnallisuus voisi olla, että uutta terveydenvaaraa luotaessa palvelimelta tarkistetaan, onko samaan tilakäyntiin liitetty samankaltaisilla tiedoilla jo terveydenvaaraa, jolloin käyttäjälle ilmoitettaisiin, että vastaavanlainen kirjaus on jo olemassa. Tällöin käyttäjä voisi valita, haluaako luoda uuden vai ei. Lisäksi vastaava tarkistus voitaisiin suorittaa jo paikallisesti ennen pyynnön lähettämistä selaimessa olevista tiedoista. Tällä lähestymistavalla vältettäisiin ylimääräinen palvelinpyyntö, mistä olisi hyötyä verkkoyhteyden ollessa heikko. Toisaalta selaimen tieto voisi olla joka tapauksessa jo vanhentunut. Toki näitä molempia tapoja voitaisiin käyttää rinnakkain tai tilanteesta riippuen. Tämä voisi myös auttaa tilanteissa, joissa sama henkilö on tehnyt eri laitteella kirjauksia, mutta ei ole onnistunut synkronoimaan muutoksiaan, jotka sitten myöhemmin synkronoitaessa tuottavat duplikaatteja toisella laitteella luotuihin kirjauksiin nähden.

Mikäli kahdella käyttäjällä on samaa terveydenvaaraa käsittelevä lomake auki samanaikaisesti, voi tämä aiheuttaa ongelmia. Tällöin molemmat muokkaavat tietoa siltä pohjalta, mikä se oli lomaketta avatessa. Silloin ei ole varmuutta tietojen olemisesta ajan tasalla lomakkeen avaamisen jälkeen. Ensimmäisen käyttäjän tiedot häviävät näin ollen, mikäli toinen käyttäjä tekee tallennuksen myöhemmin ja pyyntö myös lähtee vasta myöhemmin, sillä aikaleimakin on tällöin myöhäisempi. Tämä ongelma voitaisiin ratkaista esimerkiksi siten, että ennen päivityspyynnön lähetystä haettaisiin ensin palvelimelta uusin tieto ja tämän ollessa erilainen kuin ennen käyttäjän muutoksia, pakotettaisiin käyttäjä ratkaisemaan konflikti vastaavalla tavalla kuin aiemmin mainittiin mahdollisesti vanhentuneita tietoja päivitettäessä. Tämä muistuttaa jossain määrin versionhallintatyökaluista tuttua konfliktinratkaisua, jolloin yleensä samoihin riveihin kohdistuvat muutokset täytyy sulauttaa käsin.

Tässä sovelluksessa ei selaimeen koko sovelluksen tietokannan kopiointi ole järkevä ratkaisu, sillä offline-toiminnallisuus on vain osa sovellusta. Tästä syystä selaimeen tallennettava tieto on vain pieni osa koko sovelluksen tietokannasta eikä palvelimen ja selaimen välinen tietokannan replikoiva ratkaisu ole kovinkaan toimiva. Tällöin jouduttaisiin myös koko sovelluksen arkkitehtuuria muuttamaan toimivaksi tällaisia ratkaisuita tukevan tietokantateknologian käytön mahdollistamiseksi. Niinpä tässä sovelluksessa synkronointia ei ollut mielekästä toteuttaa esimerkiksi PouchDB:n tai CouchDB.n kaltaisilla automaattista synkronointia tukevilla menetelmillä.

6. TULOKSET JA YHTEENVETO

Tämän työn tarkoituksena oli tarkastella, kuinka web-sovelluksessa voidaan tarjota mahdollisuus tietojen tallennukseen offline-tilassa ja synkronoida sen jälkeen palvelimen kanssa. Offline-toiminnallisuuden mahdollistamiseksi tutkittiin, kuinka sovelluksen eri tyyppisiä resursseja ja dataa voidaan tallentaa verkkoselaimissa käytettäväksi. Lisäksi tarkasteltiin erilaisia synkronointimenetelmiä ja konfliktinratkaisutapoja, jotta offline-tilassa tehdyt muutokset saadaan tallennettua myös palvelimelle. Työn toteutus suoritettiin osana asiakasprojektia, johon toteutettiin offline-toiminnallisuus järjestelmässä suoritettavien tilakäyntien osalta. Lopputuloksena oli service workereita ja redux-offlinea hyödyntävä web-sovellus, jonka käyttö onnistuu ilman verkkoyhteyttä. Lisäksi tietojen synkronointi palvelimelle tapahtuu automaattisesti verkkoyhteyden ollessa jälleen saatavilla.

Sovelluksen staattisten resurssien tallentamiseen välimuistia hyödyntäen service workerit osoittautuivat hyväksi ratkaisuksi. Näiden avulla sovelluksen suorittamiseen vaadittavat tiedostot saatiin helposti tallennettua pyyntö-vastaus-pareina selaimeen, josta niitä voidaan tarjota käyttäjän saataville, mikäli verkkoyhteyttä ei ole saatavilla. Tältä osin onnistuttiin tarjoamaan mahdollisuus sovelluksen lataamiseen offline-tilassa. Tämän ansiosta sovelluksen käyttöä voidaan jatkaa myös selaimen sulkemisen jälkeen. Tähän liittyen on mahdollista optimoida jatkossa erilaisten resurssien osalta niiden tallentamista ja käyttöä, mitä ei tässä vaiheessa nähty tarpeelliseksi.

Sovelluksen datan tallentaminen ja tilan hallinta onnistuttiin toteuttamaan olemassa olevaan sovellukseen Redux-kirjastoa hyödyntäen. Lisäksi sovelluksen tilan tallentaminen ja lukeminen paikallisesti tapahtuu toteutuksessa käytettyjen ratkaisujen ansiosta automaattisesti. Lisäksi onnistuttiin offline-tilassa jonoon siirrettäviä pyyntöjä hallinnoimalla helpottamaan tietojen synkronointia palvelimelle. Myös palvelinpyyntöjen onnistumisesta riippuvat toiminnot saatiin huomioita sovelluksen tilan hallinnassa, jolloin sovelluksen tila saadaan säilytettyä palvelimen kanssa samassa tilassa. Sovelluksen tilan hallintaan ei jäänyt juurikaan paranneltavaa. Yhtenä mahdollisena kehityskohteena voi tulevaisuudessa olla tallennussijainnin vaihtaminen, mikäli tallennettavan datan määrä kasvaa.

Lopuksi tietojen synkronoinnissa saatiin aikaiseksi yksinkertainen konfliktien havaitseminen ja ratkaisu siten, että viimeisimpänä tallennetut muutokset jäävät voimaan. Tähän

pohdittiin myös vaihtoehtoisia tehokkaampia ratkaisuja, mutta niiden toteuttaminen päätettiin toistaiseksi jättää tekemättä, koska samaan alkioon liittyvien muokkausten offline-tilassa pitäisi olla sen verran harvinaisia. Jos tämä kuitenkin vaikuttaa myöhemmin muodostuvan ongelmaksi, voidaan konfliktitilanteiden ratkaisuun kehittää edistyksellisempiä keinoja, joiden avulla saadaan sulautettua eri käyttäjien tekemiä muutoksia päivityksen yhteydessä.

LÄHTEET

[1] R. T. Fielding, Architectural Styles and the Design of Network-Based Software Architectures, University of California, 2000.

[2] G. Andrews, Paradigms for process interaction in distributed programs, ACM Computing Surveys (CSUR), vol. 23, No.1, pp. 49-90, 1991.

[3] J. Tihomirovs, J. Grabis, Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics, Information Technology and Management Science, vol. 19, No.1, pp. 92-97, 2016.

[4] A. Gamble, C. Carneiro Jr, R. A. Barazi, Beginning Rails 4, (3rd ed.), Apress, 2013.

[5] M. Wasson, ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET, verkkosivu. Saatavilla (viitattu: 17.03.2019): https://msdn.microsoft.com/en-us/magazine/dn463786.aspx.

[6] S. Tilkov, Why I hate your Single Page App – freeCodeCamp.org, verkkosivu. Saatavilla (viitattu: 23.4.2019): https://medium.freecodecamp.org/why-i-hate-your-single-page-app-f08bb4ff9134.

[7] Anonymous Angular SPA: Why Single Page Applications?, verkkosivu. Saatavilla (viitattu: 23.4.2019): https://blog.angular-university.io/why-a-single-page-application-what-are-the-benefits-what-is-a-spa/.

[8] A. Syromiatnikov, D. Weyns, A journey through the land of model-view-design patterns, 2014 IEEE/IFIP Conference on Software Architecture, 2014, pp. 21-30.

[9] Anonymous ASP.NET MVC Overview | Microsoft Docs, verkkosivu. Saatavilla (viitattu: 23.4.2019): https://docs.microsoft.com/en-us/previous-versions/aspnet/dd381412(v=vs.108).

[10] TK, Understanding the basics of Ruby on Rails: HTTP, MVC, and Routes, verkkosivu. Saatavilla (viitattu: 19.03.2019): https://medium.freecodecamp.org/understanding-the-basics-of-ruby-on-rails-http-mvc-and-routes-359b8d809c7a.

[11] Anonymous A break in Laravel to understanding MVC architecture, verkkosivu. Saatavilla (viitattu: 17.03.2019): http://www.mundointerativo.com/2017/09/08/how-learn-mvc-architecture-design-patterns/.

[12] C. Gackenheimer, Introduction to React, 2015.

[13] A. Paul, A. Nalwaya, React Native for iOS Development, 2015.

[14] Anonymous Flux | Application Architecture for Building User Interfaces, Facebook Inc., verkkosivu. Saatavilla (viitattu: 14.03.2019): http://facebook.github.io/flux/docs/in-depth-overview.html#content.

[15] C. Antonio, Pro React, 2015.

[16] A. Feyerke, Designing Offline-First Web Apps, verkkosivu. Saatavilla (viitattu: 11.03.2019): https://alistapart.com/article/offline-first.

[17] P. Teixeira, Build More Reliable Web Apps with Offline-First Principles, verkkosivu. Saatavilla (viitattu: 11.03.2019): https://thenewstack.io/build-better-customer-experience-applications-using-offline-first-principles/.

[18] D. Sheppard, Beginning Progressive Web App Development: Creating a Native App Experience on the Web, 2017.

[19] C. Andreu, Offline Patterns, verkkosivu. Saatavilla (viitattu: 14.03.2019): https://www.ibm.com/developerworks/community/blogs/worklight/entry/offline_patterns?lang=en.

[20] Anonymous Cache - Web APIs | MDN, verkkosivu. Saatavilla (viitattu: 24.4.2019): https://developer.mozilla.org/en-US/docs/Web/API/Cache.

[21] J. Archibald, The Offline Cookbook, Google Developers, verkkosivu. Saatavilla (viitattu: 16.03.2019): https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/.

[22] A. Osmani and M. Cohen,Offline Storage for Progressive Web Apps, verkkosivu. Saatavilla (viitattu: 09.03.2019): https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/offline-for-pwa.

[23] Anonymous https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API, verkkosivu. Saatavilla (viitattu: 09.03.2019): https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

[24] Anonymous Can I use... Support tables for HTML5, CSS3, etc, verkkosivu. Saatavilla (viitattu: 10.03.2019): https://caniuse.com/#feat=indexeddb.

[25] B. Holt, Offline-first web and mobile apps: Top frameworks and components, verkkosivu. Saatavilla (viitattu: 11.03.2019): https://techbeacon.com/app-dev-testing/offline-first-web-mobile-apps-top-frameworks-components.

[26] Ł Huculak, Offline-first application architecture, verkkosivu. Saatavilla (viitattu: 17.03.2019): https://codete.com/blog/offline-first-application-architecture/.

[27] P. Teixeira, How to build a reliable transaction experience for your customers, verkkosivu. Saatavilla (viitattu: 17.03.2019): https://medium.com/yld-engineering-blog/how-to-build-a-reliable-transaction-experience-for-your-customers-b082149adfed.

[28] M. Faiz, U. Shanker, Data synchronization in distributed client-server applications, 2016 IEEE International Conference on Engineering and Technology (ICETECH), 2016, pp. 611-616.

[29] M. Hopson, How to create a Rails project with a React and Redux front-end (plus Typescript!), verkkosivu. Saatavilla (viitattu: 21.03.2019): https://medium.freecodecamp.org/how-to-create-a-rails-project-with-a-react-and-redux-front-end-8b01e17a1db.

[30] A. M. Vipul, P. Sonpatki, ReactJS by Example - Building Modern Web Applications with React, 2016.

[31] Anonymous State and Lifecycle – React, verkkosivu. Saatavilla (viitattu: 16.04.2019): https://reactjs.org/docs/state-and-lifecycle.html.

[32] Anonymous Usage with React · Redux, verkkosivu. Saatavilla (viitattu: 16.04.2019): https://redux.js.org/basics/usage-with-react.

[33] Anonymous Using Service Workers, verkkosivu. Saatavilla (viitattu: 09.03.2019): https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[34] Anonymous GitHub - rails/webpacker: Use Webpack to manage app-like JavaScript modules in Rails, verkkosivu. Saatavilla (viitattu: 13.04.2019): https://github.com/rails/webpacker.

[35] Anonymous The Asset Pipeline — Ruby on Rails Guides, verkkosivu. Saatavilla (viitattu: 13.04.2019): https://guides.rubyonrails.org/asset_pipeline.html.

[36] Anonymous What Web Can Do Today, verkkosivu. Saatavilla (viitattu: 23.04.2019): https://whatwebcando.today/offline.html.

[37] Anonymous Using the application cache - HTML: Hypertext Markup Language | MDN, verkkosivu. Saatavilla (viitattu: 13.04.2019): https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache.

[38] Anonymous GitHub - NekR/offline-plugin: Offline plugin (ServiceWorker, AppCache) for webpack (https://webpack.js.org/), verkkosivu. Saatavilla (viitattu: 16.04.2019): https://github.com/NekR/offline-plugin.

[39] J. Eväkallio, Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native, verkkosivu. Saatavilla (viitattu: 11.03.2019): https://hackernoon.com/introducing-redux-offline-offline-first-architecture-for-progressive-web-applications-and-react-68c5167ecfe0.

[40] Anonymous Redux Offline – Build Offline-First Apps for Web and React Native, verkkosivu. Saatavilla (viitattu: 14.04.2019): https://www.ctolib.com/topics-112916.html.

[41] Anonymous Managing HTML5 Offline Storage - Google Chrome, verkkosivu. Saatavilla (viitattu: 16.04.2019): https://developer.chrome.com/apps/offline_storage.

[42] Anonymous tools/packages/smart-queue at master · redux-offline/tools · GitHub https://github.com/redux-offline/tools/tree/master/packages/smart-queue.

LIITE A: SERVICE WORKERIN ASENNUS

```
self.addEventListener('install', function(event) {
  // Staattisten resurssien versionhallintaan käytettävä versionumero.
  var CACHE_VERSION = 'v1';
  var CACHE_NAME = CACHE_VERSION + ':mela-tth';

  // Tämän avulla saadaan päivitettyä versio uuteen,
  // vaikka aikaisemmin asennettu service worker olisi vielä aktiivinen.
  self.skipWaiting();

  event.waitUntil(
   caches.open(CACHE_NAME).then(function prefill(cache) {
     return cache.addAll([
       // Asset pipelinen sisältämä JavaScript
       '<%= asset_path "application.js" %>',
       // Webpackerin paketoima käyttöliittymäsovellus
       '<%= asset_pack_path "application.js" %>',

       // Kuvat
       '<%= asset_path "background.jpg" %>',
       '<%= asset_path "mela_logo.png" %>',

       // Fontit
       '<%= asset_path "font-awesome/fa-solid-900.eot" %>',
       '<%= asset_path "font-awesome/fa-solid-900.woff2" %>',
       '<%= asset_path "font-awesome/fa-solid-900.woff" %>',
       '<%= asset_path "font-awesome/fa-solid-900.ttf" %>',
       '<%= asset_path "font-awesome/fa-solid-900.svg" %>',

       '<%= asset_path "fonts/adalicons.eot" %>',
       '<%= asset_path "fonts/adalicons.woff" %>',
       '<%= asset_path "fonts/adalicons.ttf" %>',
       '<%= asset_path "fonts/adalicons.svg" %>',

       // Sovelluksen tyylit
       '<%= asset_path "application.css" %>',

       // Käännökset
       '<%= translations_path %>',
       // HTML-pohja
       '<%= offline_layout_path %>'
     ]);
   })
 );
}
```

LIITE B: PYYNTÖJEN LISÄÄMINEN JONOON

```
// Muokattu tästä https://github.com/redux-offline/tools/blob/mas-
ter/packages/smart-queue/enqueue.js

export function enqueue(array, action, context) {
 const outbox = array;
 let queueAction;
 const {
   meta: {
     offline: { queue = null }
   }
 } = action;

 if (!queue) {
   // Oletusarvoinen enqueue-metodin toiminnallisuus
   return [...outbox, action];
 }


 const { method = 'UNKNOWN', key = null, scope = 'app', isTempId =
false } = queue;

 validate(key);
 // Haetaan pyyntöjä samalla tunnisteella
 const index = indexOfAction(outbox, key, scope);

 switch (method) {
   case CREATE:
     return [...outbox, action];
   case DELETE:
     if (index !== -1) {
       queueAction = outbox[index];
       if (
         safeToProceed(index, context) &&
         (queueAction.meta.offline.queue.method === UPDATE ||
           queueAction.meta.offline.queue.method === CREATE)
       ) {
         outbox.splice(index, 1);
       }
     }
     if (isTempId) {
       // Väliaikaisiin tunnisteisiin liittyviä pyyntöjä ei lähetetä
       return outbox;
     }
     return [...outbox, action];
```

```
    case UPDATE:
      if (index !== -1) {
        queueAction = outbox[index];
        if (safeToProceed(index, context)) {
          if (queueAction.meta.offline.queue.method === CREATE) {
            outbox[index] = mergeUpdateToCreate(outbox[index], action);
            return outbox;
          } else if (queueAction.meta.offline.queue.method === UPDATE)
{
            outbox[index] = mergeActions(outbox[index], action);
            return outbox;
          }
        }
      }
      if (isTempId) {
        // Väliaikaisiin tunnisteisiin liittyviä pyyntöjä ei lähetetä
        return outbox;
      }
      return [...outbox, action];
    case READ:
     // Lukuoperaatiot suoritetaan muista operaatiosta erillisessä nä-
kyvyysalueessa, koska ne eivät muuta sovelluksen tilaa.
     // Tällöin niiden olemassaolo ei vaikuta kirjoitusoperaatioiden
sulauttamiseen.
     if (index !== -1) {
        queueAction = outbox[index];
        if (
          safeToProceed(index, context) &&
          queueAction.meta.offline.queue.method === READ
        ) {
          outbox[index] = mergeActions(outbox[index], action);
        }
      } else {
        return [...outbox, action];
      }
      return outbox;
    default:
      throw new Error(
        'Missing method definition, the "method" value should be either
of [CREATE, READ, DELETE, UPDATE]!'
      );
  }
}
```